pyramid_sqla Documentation

Release 2.0b1

Mike Orr

January 03, 2012

CONTENTS

Version 2.0b1, released XXXX-XX-XX

PyPI http://pypi.python.org/pypi/Akhet

Docs http://docs.pylonsproject.org/projects/akhet/dev/

Source https://bitbucket.org/sluggo/akhet (Mercurial)

Bugs https://bitbucket.org/sluggo/akhet/issues

Discuss pylons-discuss list

Akhet is a set of tutorial-level documentation and convenience code for Pyramid. Version 2 focuses more heavily on documentation, and does not contain an application scaffold [1]. Instead the documentation shows how to customize Pyramid's built-in scaffolds to give a Pylons-like environment. The documentation walks through the default application's structure, providing a supplement to the Pyramid documentation and showing how the structure differs from Pylons. It also discusses some alternative APIs and the tradeoffs between them. The Akhet library (the convenience classes) are unchanged in this release.

Akhet 2.0 runs on Python 2.5 - 2.7. The next version will probably add Python 3 and drop Python 2.5, as Pyramid 1.3 is doing.

[1] The term "scaffold" is the same as "application template", "paster template", and "skeleton". Pyramid has standardized on the term "scaffold" to avoid confusion with HTML templates.

CHAPTER

ONE

INTRODUCTION TO AKHET 2

This chapter recounts Akhet's history and current status. The next chapter introduces some vocabulary terms. The third chapter summarizes how to create a Pyramid application using the recommended skeleton, and how to use the Pyramid and Akhet development versions. The subsequent chapters analyze different aspects of the default application, and discuss various enhancements you can add to it. The final chapters discuss other Pyramid topics.

Akhet evolved out of a Pyramid/SQLAlchemy application scaffold. It then grew more Pylons-like features, a small library, and documentation that expanded to become a general introduction to Pyramid. In Akhet 2, the documentation takes center stage and the scaffold has been retired. Why this reversal? Mainly because it's so much work to maintain a scaffold. (The scaffold rant appendix has the full details.) Also, Pyramid 1.3 consolidates the built-in scaffolds to three well-chosen ones.

Routing mechanism	Database	Pyramid 1.3 scaffold	Pyramid 1.2.4 scaffold
URL dispatch	SQLAlchemy	**alchemy**	routesalchemy
URL dispatch	-	**starter**	-
Traversal	ZODB	**zodb**	zodb
Traversal	SQLAlchemy	-	alchemy
Traversal	-	-	starter

The Pyramid 1.3 scaffolds emphasize on the two most widely-used application styles: URL dispatch with SQLAlchemy, and traversal with ZODB. The scaffold names are simplified to focus on this goal. The 'starter' scaffold switches to URL dispatch because it's more appropriate for beginners. Using traversal at all is an advanced topic, and especially with SQLAlchemy. (If you want to see how traversal with SQLAlchemy can be implemented robustly, check out the Kotti content-management system.)

For those coming from Akhet 1, Pylons, Django, Rails, and similar frameworks with something akin to Routes, URL dispatch and SQL databases will be somewhat familiar. For those coming from a Java servlet or PHP-without-a-framework background, URL dispatch will be new but it's a good rule-based way to handle URLs. Traversal is an entirely different concept, which is most useful when site users are allowed to create their own URLs with multiple levels (as in a content-management system or file manager). Traversal maps naturally to nested objects, which is why it's often paired with an object database.

So this Akhet book and the Pyramid developers both recommend that new users start with the 'alchemy' or 'starter' scaffold in Pyramid 1.3, or the 'routesalchemy' scaffold in Pyramid 1.2. The rest of this book is based on those scaffolds.

Two other changes in Pyramid 1.3 deserve mention. One, it's compatible with Python 3, and it drops Python 2.5 support. Akhet 2.0 does not do either of these yet, but the next version probably will.

Two, as part of the Python 3 porting, Pyramid 1.3 dropped its Paste and PasteScript dependencies. These will probably not be ported to 3 for reasons listed in the scaffold rant. This has the following consequences:

- The 'paster' command is gone. It's replaced by 'pcreate' and 'pserve'.

- The default HTTP server is now Wsgiref, the one in the Python standard library. You can use it during development and switch to a more robust server for production (PasteHTTPServer, CherryPy, mod_wsgi, FastCGI, etc).
- PasteDeploy is <i>not</i> dropped, so the INI files still work the same way.

CHAPTER

TWO

PYRAMID VOCABULARY

This is a supplement to the Glossary in the Pyramid manual. It focuses on the subset of terms critical for Akhet, and compares them to Pylons.

Router

A Pyramid WSGI application, which is an instance of `pyramid.router.Router`. Equivalent to `PylonsApp`.

View (View Callable)

A function or method equivalent to a Pylons controller action. It takes a `Request` object representing a web request, and returns a `Response` object. (The return value may be different when using a *renderer*.)

View Class (Handler)

A class containing view methods, so equivalent to a Pylons controller. If the view is a method rather than a function, the `request` argument is passed to the class constructor rather than to the method. The `pyramid_handlers` package offers one Pylons-like way to define and register view classes.

MVC

The Model-View-Controller pattern used in programming. Pyramid is more of a MV (Model-View) framework than MVC. Many contemporary web developers have given up on MVC as not being well suited to the web. MVC envisions a three-way split between business logic, user interface, and framework interface, but in practice the latter two are hard to separate. A two-way split is more useful: the *model* which is all code specific to your business and can be used on its own, and the *view* which is all code specific to the framework, user interface, and HTTP/HTML environment. (The "controller" then is the framework itself.)

URL Dispatch

A routing mechanism similar to Routes. It chooses a view for each incoming URL based on a set of rules.

Traversal

Pyramid's other routing mechanism, which maps the URL's components to a recursive object-oriented data structure. Traversal is an advanced topic; beginners are advised to stick to URL dispatch at first. Traversal is useful mainly in applications that allow users to define arbitrary URL subtrees, such as a content-management system (CMS) or a web-based file manager. URL dispatch, in contrast, works better when the URLs are known ahead of time or when they're a fixed depth (e.g., "/articles/{id}").

If no route matches the URL, Pyramid tries traversal as a fallback. The data structure is null by default, so this is a no-op.

Context

An object accessible to the view, which tells the "context" it was invoked in. This does not exist in Pylons. It's an additional piece of information distinct from the routing variables, query parameters, and other aspects of the request. It plays an important role in traversal, and in some advanced usages of URL dispatch.

Resource Tree, Root, Resource, Root Factory

These are all used in traversal, and in some advanced usages of URL dispatch.

A *resource tree* is the data structure that traversal maps the URL to. It's a recursive dict-like structure. The top-level node is called the *root*. A *root factory* is a callable that returns a *root*; i.e., the top node of a live resource tree. The *resource tree* is most commonly a ZODB database, but it can also be implemented in SQL or on-the-fly (by a root object with a clever `.__getitem__` method that creates child nodes on demand).

If the request URL is "/a/b/c", traversal maps it to `root["a"]["b"]["c"]`. The final node (i.e., the value of the "c" key) is called the *resource*. That object is delivered to the view as the *context*.

In URL dispatch, a *root factory* is normally not specified, and it defaults to a null factory which causes the *context* to be `None`. However, you can specify a custom *root factory* at either the top level or on an individual route. In this case, the factory should return a *resource* which will **be** the *context*.

Request

An object which contains all state data pertinent to the current web request and the application runtime. It's a subclass of `WebOb.Request`. Its attributes subsume the functionality of several Pylons globals (request, response, session, tmpl_context or c, url), the match dict, query parameters, etc.

Response

An object which specifies what kind of response to return to the user: the HTTP status, HTTP headers, and body content. It's normally a subclass of `WebOb.Response` but you can substite any object with the appropriate `status`, `headerlist`, and `app_iter` attributes. A view must return a Response unless it's using a renderer.

Renderer

A function that takes a view's return value as input, and returns a Response. Normally the view returns a dict of data values, and the renderer invokes a template to produce the Response body. Some renderers instead serialize the dict into another format such as JSON.

Event, Subscriber

A mechanism for running arbitrary code at specific points during request processing or during the application's lifetime. You register *subscriber* callbacks for specific events, and Pyramid will call those callbacks when those events happen. The callback's arguments allow access to pertinent state data.

Asset Spec

A fully qualified Python module name or object name, such as the strings "myapp.handlers" or "myapp.handlers:MyHandler". Many Pyramid methods accept these as arguments in lieu of the actual object. The colon separates the last item to import (a package or module) from the first item to fetch via attribute access (a variable in the module).

Certain methods require an asset spec pointing to a non-Python file or directory inside a Python package. In this case, the right side of the colon is the relative path inside the package, using "/" delimeters regardless of platform. For instance, "myapp:static/" or "myapp.lib:images/logo.png".

Settings

A dict of application configuration settings. This combines:

- “deployment settings” parsed from the INI file (or passed in by whatever top-level script launches the application).
- “application settings”, or site-wide constants, set in the main function.
- “application globals”: data structures, non-SQLAlchemy database connections, a cache object, or other things that are global to all requests. These are also normally set in the main function.

Registry

An object that is global to the application and contains internal framework data such as which routes and views have been defined. Application writers generally ignore it except when they need a setting, which are in its `.settings` attribute.

CHAPTER

THREE

INSTALLING PYRAMID AND CREATING APPLICATIONS

Here are the basic steps to install Pyramid and Akhet and create an application. For more details see the Installing Pyramid and Creating a Pyramid Project chapters in the Pyramid manual. New users should also do the SQLAlchemy + URL Dispatch Wiki Tutorial (Wiki2), which explains Pyramid while you build a simple wiki application. The following chapters will walk through this application. For convenience, a prebuilt tarball of the application is available: wiki2-tutorial.tar.gz [1].

The steps here are effectively the same as the installation chapter of the Wiki tutorial; we're just using pip rather than other installation commands because it makes uninstallation easier, and because it's the new hotness. We also activate the virtualenv, which allows us to keep the application source outside the virtualenv without having to type convoluted paths to run virtualenv commands. Keeping the application outside the virtualenv makes it easier to delete/recreate the virtualenv if it gets hosed, and to run the application under multiple virtualenvs (e.g., to see how it works under different Python versions, different Pyramid versions, and different dependency versions).

These steps assume you have Python, virtualenv, and SQLite installed.

3.1 Creating an application with Pyramid 1.3 and Akhet

(Pyramid 1.3 is unreleased as of this writing. This alternative won't work until it's released, so use one of the other two alternatives instead.)

```
$ virtualenv --no-site-packages ~/directory/myvenv
$ source ~/directory/myvenv/bin/activate
(myvenv)$ pip install 'Pyramid>=1.3'
(myvenv)$ pip install Akhet
(myvenv)$ pcreate -s alchemy Zzz
(myenv)$ cd Zzz
(myenv)$ pip install -e .
(myenv)$ populate_Zzz development.ini
(myenv)$ pserve development.ini
```

[1] Copied from the Pyramid repository, directory *docs/tutorials/wiki2/src/tests* (renamed 'tests' directory to 'wiki2-tutorial'). Revision dated 2011-12-08, ID 674636494b7e546598ac3adb094c3dca6f6b8c9e.

3.2 Creating an application with Pyramid 1.2 and Akhet

```
$ virtualenv --no-site-packages ~/directory/myvenv
$ source ~/directory/myvenv/bin/activate
(myvenv)$ pip install 'Pyramid<1.3'
(myvenv)$ pip install Akhet
(myvenv)$ paster create -t routesalchemy Zzz
(myenv)$ cd Zzz
(myenv)$ pip install -e .
(myenv)$ populate_Zzz development.ini
(myenv)$ paster serve development.ini
```

3.3 Creating an application with development versions of Pyramid and Akhet

Installing Akhet from its source repository works like most Python repositories. Pyramid, however, requires additional steps.

```
$ virtualenv --no-site-packages ~/directory/myvenv
$ source ~/directory/myvenv/bin/activate
(myvenv)$ git clone git://github.com/Pylons/pyramid Pyramid
(myvenv)$ pip install setuptools-git
(myvenv)$ pip install -e ./Pyramid
(myvenv)$ hg clone http://bitbucket.org/sluggo/akhet Akhet
(myvenv)$ pip install -e ./Akhet
(myvenv)$ pcreate -s alchemy Zzz
(myenv)$ cd Zzz
(myenv)$ pip install -e .
(myenv)$ populate_Zzz development.ini
(myenv)$ pserve development.ini
```

Three things to note here:

- Install Pyramid first so that it will satisfy Akhet's Pyramid dependency. Otherwise when you install Akhet, it will download the latest stable version of Pyramid from PyPI.
- Pyramid requires 'setuptools-git' because the repsository contains Git submodules.
- Pyramid *must* be installed as a link (with "-e") because a regular install won't copy the scaffolds or other supplemental files. (That's because the repository does not contain a MANIFEST.in file.)
- The extra "./" is so that "pip install -e" recognizes the argument as a path name rather than as something to download from PyPI.

3.4 Observations

The "–no-site-packages" option is recommended for Pyramid; it isolates the virtualenv from packages installed globally on the computer, which may be incompatible or have conflicting versions. If you have trouble installing a package that has C extensions (e.g., a database library, PIL, NumPy), you can try making a symlink from the virtualenv's site-packages directory to the OS version of the package; it may take some jiggering to make the package happy.

I found –no-site-packages necessary on Ubuntu 10 because Ubuntu installs some Zope packages but not all the ones Pyramid needs, and `zope` is a namespace package which can't be split between the global directory and the virtualenv. I have not had this problem with Ubuntu 11.10 so far, so it may be fixed.

"pip install -e ." installs the application and all dependencies listed in setup.py. That's necessary for this application because it depends on SQLAlchemy, which is not installed with raw Pyramid. Installation also sets up the 'populate_Zzz' command. In a simpler application without these restrictions (such as the 'starter' scaffold), you can get by without installation. You'll have to run "python setup.py egg_info" instead (which updates the distribution's metadata, and is one of the installation steps. Also, if you don't install the application, you'll have to always chdir to the application's directory before running it, because Python won't be able to import it otherwise.

Remember for later: whenever you add or delete a file in the application directory, run "python setup.py egg_info" to update the metadata.

See Uninstalling if you want to uninstall things later.

3.5 Uninstalling

To uninstall an application or package that was installed via pip, use "pip uninstall":

```
(myvenv)$ pip uninstall Zzz
```

If you installed it via "easy_install", "python setup.py install", or "python setup.py develop", you'll have to uninstall it manually. Chdir to the virtualenv's *site-packages* directory. Delete any subdirectories and files corresponding to the Python package, its metadata, or its egg link. For our sample application these would be *zzz* (Python package), *Zzz.egg-info* (pip egg_info), *Zzz.egg* (easy_install directory or ZIP file), and *Zzz.egg-link* (egg link file). Also edit *easy-install.pth* and delete the application's line if present.

The tarball was built with Pyramid 1.3-dev (2011-12-02, rev. d5666e630a08c943a22682540aa51174cee6851f), Python 2.7.2, on Ubuntu 11.10 (Linux).

CHAPTER

FOUR

PYRAMID ARCHITECTURE

This chapter walks through the default 'alchemy' application you created in the last chapter (or downloaded the tarball). We'll skim just briefly over material that's covered in the Wiki tutorial, instead focusing on how Pyramid differs from Pylons.

The Zzz application has several aspects similar to Pylons: INI files, a startup function, views (called Controllers in Pylons), templates, and models. However, the filenames are different and the API syntax is different. Pyramid is more flexible than Pylons, so you can create a minimal application in a single Python module, and run it in the same module without an INI file or "pserve". However, we'll stick to the 'alchemy' scaffold which creates a directory structure scalable to large applications.

4.1 Directory layout

The default application contains the following files after you install it:

```
Zzz
-- CHANGES.txt
-- MANIFEST.in
-- README.txt
-- development.ini
-- production.ini
-- setup.cfg
-- setup.py
-- zzz
|   -- __init__.py
|   -- models.py
|   -- scripts
|   |   -- __init__.py
|   |   -- populate.py
|   -- static
|   |   -- favicon.ico
|   |   -- footerbg.png
|   |   -- headerbg.png
|   |   -- ie6.css
|   |   -- middlebg.png
|   |   -- pylons.css
|   |   -- pyramid.png
|   |   -- pyramid-small.png
|   |   -- transparent.gif
|   -- templates
|   |   -- mytemplate.pt
|   -- tests.py
|   -- views.py
-- Zzz.egg-info
    -- dependency_links.txt
    -- entry_points.txt
    -- not-zip-safe
    -- PKG-INFO
    -- requires.txt
    -- SOURCES.txt
    -- top_level.txt
```

4.2 development.ini

development.ini has the same structure as Pylons, and it's actually simpler than earlier versions of Pyramid. The application and server sections look like this:

```
[app:main]
use = egg:Zzz

pyramid.reload_templates = true
pyramid.debug_authorization = false
pyramid.debug_notfound = false
pyramid.debug_routematch = false
pyramid.debug_templates = true
pyramid.default_locale_name = en
pyramid.includes = pyramid_debugtoolbar
                   pyramid_tm

sqlalchemy.url = sqlite:///%(here)s/Zzz.db

[server:main]
use = egg:pyramid#wsgiref
host = 0.0.0.0
port = 6543
```

The logging sections will be covered in the logging chapter. Differences between development.ini and production.ini will be covered in the deployment chapter.

When you run "pserve development.ini", it does the following:

1. Activate logging based on the logging sections.
2. Read the "[app:main]" section and instantiate the specified application.
3. Read the "[server:main]" section and instantiate the specified server.
4. Launch the server with the application, and let it process requests forever.

In the app section, the "use = egg:Zzz" tells which Python distribution to load, in this case our "Zzz" application. The "pyramid.*" options are mostly debugging variables. Set any of them to "true" to enable various kinds of debug logging. "pyramid.default_locale_name" sets the predominent region/language for the application. "pyramid.reload_templates" tells whether to recheck the timestamp of template source files whenever it renders a template, in case the file has been updated since startup. (Mako and Chameleon respect this value, but not all template engines understand it.)

"pyramid_includes" specifies optional "tweens" to wrap around the application. Tweens are like WSGI middleware but are specific to Pyramid. In other words, they're generic services that can be wrapped around a variety of applications. "pyramid_debugtoolbar" is the debug toolbar at the right margin of browser screens, and shows an interactive traceback screen if an exception occurs. "pyramid_tm" is the transaction manager, which is covered in a later chapter.

(To see the interactive traceback in action, skip the "populate_Zzz" step or delete the "Zzz.db" file, and run pserve. It will error out because a database table doesn't exist. If it doesn't give an error, add a `raise RuntimeError` line in the `my_view` function in *zzz/views.py*.)

"sqlalchemy.url" tells which database the application should use. "%(here)s" expands to the path of the directory containing the INI file.

The "[server:main]" section is the same as in Pylons. It tells which WSGI server to run. Pyramid 1.3 defaults to the wsgiref HTTP server in the Python standard library. It's single-threaded so it can only handle one request at a time, but that's good enough for development or debugging.

Pyramid no longer uses WSGI middleware by default. If you want to add your own middleware, see the PasteDeploy manual for the syntax. But first consider whether making a Pyramid tween would be more convenient.

4.2.1 Init module and main function

A Pyramid application revolves around a top-level `main()` function in the application package. "pserve" does the equivalent of this:

```
# Instantiate your WSGI application
import zzz
app = zzz.main(**settings)
```

The Pylons equivalent to `main()` is `make_app()` in middleware.py. The `main()` function replaces Pylons' middleware.py, config.py, *and* routing.py but is much shorter:

```
1  from pyramid.config import Configurator
2  from sqlalchemy import engine_from_config
3
4  from .models import DBSession
5
6  def main(global_config, XXsettings):
7      """ This function returns a Pyramid WSGI application.
8      """
9      engine = engine_from_config(settings, 'sqlalchemy.')
10     DBSession.configure(bind=engine)
11     config = Configurator(settings=settings)
12     config.add_static_view('static', 'static', cache_max_age=3600)
13     config.add_route('home', '/')
14     config.scan()
15     return config.make_wsgi_app()
```

(*Doc limitation*: `XXsettings` in line 6 is actually `**settings`. We had to alter it in the docs to prevent Vim's syntax highlighting from going bezerk.)

This main function is short and sweet. Later we'll discuss lots of things you can add here to add features.

"pserve" parses the "[app:main]" options into a dict called "settings". It calls the `main()` function with the settings as keyword args. (The `global_config` arg is not used much; it's covered later.)

Lines 9-10 instantiate a database engine based on the "sqlalchemy." settings in the INI file. `DBSession` is a global object used to access SQLAlchemy's object-relational mapper (ORM).

Line 11 instantiates a `Configurator` which will instantiate the application. (It's not the application itself.)

Line 12 tells the configurator to serve the static directory (*zzz/static*) under URL "/static". The arguments are more than they appear, as we'll see in the customization section.

Line 13 creates a route for the home page. This is more or less like a Pylons route, except it doesn't specify a controller and action.

Line 14 scans the application's Python modules looking for views to register. This imports all the modules under `zzz`.

Line 15 instantiates a Pyramid WSGI application based on the configuration, and returns it.

4.3 Configurator constructor arguments

The Configurator constructor accepts the following optional arguments:

authentication_policy, authorization_policy

> Callbacks to enable Pyramid's built-in authorization mechanism. The Authorizatoin chapter in the Wiki tutorial has an example of their use.

root_factory

> A callback that returns a *resource root*. See next section. has an extra piece of data called the *context*. this, so Pyramid uses a default factory that always returns `None`. value called the *context* the *context* deliveredc to the view as its *context*. In URL dispatch, the root will be delivered directly to the view as the *context*. The example does *not* specify this argument, so Pyramid uses a default root factory that always delivers `None` as a context. The Wiki tutorial

Note that the `Configurator` constructor does *not* have a `root_factory=` argument. In URL dispatch, root factories are used only in advanced cases. The argument is a mainly to set authorization permissions or as a fancy way to deliver database objects to the view. mainly to set up authorization permissions or to deliver database objects to the view. Pyramid will fall back to a default root factory, which always delivers a `None` context to the view.

4.4 Route arguments and predicates

`config.add_route` accepts a large number of keyword arguments. Here are the ones most commonly used in Pylons-like applications.

The arguments are divided into *predicate arguments* and *non-predicate arguments*. Predicate arguments determine whether the route matches the current request: all predicates must pass in order for the route to be chosen. Non-predicate arguments do not affect whether the route matches.

name

> [Non-predicate] The first positional arg; required. This must be a unique name for the route. The name will be used to register a view for this route, and to generate URLs.

pattern

> [Predicate] The second positional arg; required. This is the URL path with optional "{variable}" placeholders; e.g., "/articles/{id}" or "/abc/{filename}.html". The leading slash is optional. By default the placeholder matches all characters up to a slash, but you can specify a regex to make it match less (e.g., "{variable:d+}" for a numeric variable) or more ("{variable:.*}" to match the entire rest of the URL including slashes). The substrings matched by the placeholders will be available as *request.matchdict* in the view.
>
> A wildcard syntax "*varname" matches the rest of the URL and puts it into the matchdict as a tuple of segments instead of a single string. So a pattern "/foo/{action}/*fizzle" would match a URL "/foo/edit/a/1" and produce a matchdict `{'action': u'edit', 'fizzle': (u'a', u'1')}`.
>
> Two special wildcards exist, "*traverse" and "*subpath". These are used in advanced cases to do traversal on the remainder of the URL.

factory

[Non-predicate] A callable (or asset spec). This is used to define a route-specific context. In URL dispatch, this returns a *root resource* which is also used as the *context*. If you don't specify this, a default root will be used. In traversal, the root contains one or more resources, and one of them will be chosen as the context.

xhr

[Predicate] True if the request must have an "X-Reqested-With" header. Some Javascript libraries (JQuery, Prototype, etc) set this header in AJAX requests.

request_method

[Predicate] An HTTP method: "GET", "POST", "HEAD", "DELETE", "PUT". Only requests of this type will match the route.

path_info

[Predicate] A regex compared to the URL path (the part of the URL after the application prefix but before the query string). The URL must match this regex in order for the route to match the request.

request_param

[Predicate] If the value doesn't contain "=" (e.g., "q"), the request must have the specified parameter (a GET or POST variable). If it does contain "=" (e.g., "name=value"), the parameter must have the specified value.

This is especially useful when tunnelling other HTTP methods via POST. Web browsers can't submit a PUT or DELETE method via a form, so it's customary to use POST and to set a parameter `_method="PUT"`. The framework or application sees the "_method" parameter and pretends the other HTTP method was requested. In Pyramid you can do this with `request_param="_method=PUT`.

header

[Predicate] If the value doesn't contain ":"; it specifies an HTTP header which must be present in the request (e.g., "If-Modified-Since"). If it does contain ":", the right side is a regex which the header value must match; e.g., "User-Agent:Mozilla/.*". The header name is case insensitive.

accept

[Predicate] A MIME type such as "text/plain", or a wildcard MIME type with a star on the right side ("text/*") or two stars ("*/*"). The request must have an "Accept:" header containing a matching MIME type.

custom_predicates

[Predicate] A sequence of callables which will be called in order to determine whether the route matches the request. The callables should return `True` or `False`. If any callable returns `False`, the route will not match the request. The callables are called with two arguments, `info` and `request`. `request` is the current request. `info` is a dict which contains the following:

```
info["match"]  =>  the match dict for the current route
info["route"].name  =>  the name of the current route
info["route"].pattern  =>  the URL pattern of the current route
```

Use custom predicates argument when none of the other predicate args fit your situation. See <http://docs.pylonsproject.org/projects/pyramid/1.0/narr/urldispatch.html#custom-route-predicates>` in the Pyramid manual for examples.

You can modify the match dict to affect how the view will see it. For instance, you can look up a model object based on its ID and put the object in the match dict under another key. If the record is not found in the model, you can return False to prevent the route from matching the request; this will ultimately case HTTPNotFound if no other route or traversal matches the URL. The difference between doing this and returning HTTPNotFound in the view is that in the latter case the following routes and traversal will never be consulted. That may or may not be an advantage depending on your application.

4.4.1 Models

This is where you define your domain model; i.e., what makes this application different from other Pyramid applications. A good application structure separates domain logic (not Pyramid-specific or UI-related) from view logic (Pyramid-specific or UI-related). This makes it easy to use the domain code outside of the web application (in standalone utilities) or to port it to another framework (if you ever decide to do so).

Note: the term "model" is used in two different ways. Collectively it means all your ORM classes and domain logic together. (One model per application.) Individually it means a single ORM class. (Several models in one application.) Either way is fine, but beware that the word "model" (singular) can mean one or the other. This led to a controversy in both Pylons and Pyramid on whether to put "model" or "models" in the scaffolds. Pylons chose "model"; Pyramid chose "models". But it doesn't matter, and you can rename models.py to model.py if you wish. Just be aware that the word "model" can mean either one class or all classes together.

At minimum you should define your database tables and ORM classes here. Some people also put other business logic here, either as methods in the ORM classes or as functions. Other people put miscellaneous domain logic into a 'lib' package (*zzz/lib*). Others put the entire models and domain logic in a separate Python distribution, which they import into the Pyramid application. Others put domain logic directly into the views, but this is not recommended unless it's a small amount of code because it mixes framework-independent and framework-dependent code.

The default *zzz/models.py* looks like this:

```
from sqlalchemy import (
    Column,
    Integer,
    Text,
    )

from sqlalchemy.ext.declarative import declarative_base

from sqlalchemy.orm import (
    scoped_session,
    sessionmaker,
    )

from zope.sqlalchemy import ZopeTransactionExtension

DBSession = scoped_session(sessionmaker(extension=ZopeTransactionExtension()))
Base = declarative_base()

class MyModel(Base):
    __tablename__ = 'models'
    id = Column(Integer, primary_key=True)
    name = Column(Text, unique=True)
    value = Column(Integer)

    def __init__(self, name, value):
        self.name = name
        self.value = value
```

This is just one way to organize a SQLAlchemy model. All SQLAlchemy models models will have a DBSession, declarative base, and ORM classes unless you're doing something unusual like not using the object-relational mapper or not using the declararive syntax. But different people have different ideas about what to name the variables or where to put them. We'll see an alternative structure later. If you're unfamiliar with SQLAlchemy, the SQLAlchemy manual is well written and informative.

4.4.2 Views

A Pyramid view is equivalent to a Pylons controller action: it's the page-specific routine where the request is processed and an HTML response is generated. The default *zzz.views* module looks like this:

```python
from pyramid.view import view_config

from .models import (
    DBSession,
    MyModel,
    )

@view_config(route_name='home', renderer='templates/mytemplate.pt')
def my_view(request):
    one = DBSession.query(MyModel).filter(MyModel.name=='one').first()
    return {'one':one, 'project':'Zzz'}
```

A view can be either a function or a method, and it can be located anywhere. You must register each view with the configurator, either by calling `config.add_view()` or – as in this example – by using the `@view_config` decorator in conjunction with `config.scan()`. The scan method imports the application's modules recursively, and registers all callables decorated with `@view_config`.

Pylons is more specific about actions. An action must be a method in a controller class, the class must inherit from a compatible base class, and the class must be located in a module determined by the class's name. The action may take arguments corresponding to routing variables, and it normally ends with "return render(TEMPLATE_NAME)", which renders an HTML string. Magic global variables contain the current request, response, session, cache, application globals, and a URL generator.

A Pyramid view *function* takes a `Request` and returns a `Response`. But if the `@view_config` has a `renderer=` argument, it returns a dict of template variables instead. The renderer invokes the specified template with the variables, and generates the `Response`. This is similar to TurboGears. It has advantages in unit testing and with alternate output formats (such as JSON). That's because `@view_config` *does not change the function's arguments or return value*. It merely sets function attributes which affect how the view is registered. So a unit test that calls the view function directly gets back the dict of variables, not a HTML string.

In the example, the `renderer=` arg names a template. Because the filename ends in ".pt", Pyramid recognizes it as a Chameleon template and invokes the Chameleon renderer. The renderer fills the template with the variables and generates a `Response`. A Mako renderer is also available, and non-template renderers for JSON and other formats. The view's return value is whatever type the renderer accepts. Template renderers normally accept dicts, while non-template renderers may accept other types.

A view *method* is like a view function, except that the `request` argument is passed to the class's constructor rather than to the method. So a view class is normally defined like this:

```python
class MyViewClass(object):
    def __init__(self, request):
        self.request = request

    @view_config(route_name='home', renderer='templates/mytemplate.pt')
    def my_view(self):
        # req = self.request
        one = DBSession.query(MyModel).filter(MyModel.name=='one').first()
        return {'one':one, 'project':'Zzz'}
```

Later we'll see a `Handler` base class that takes this a step further.

The `route_name=` argument to `@view_config` tells which route to attach this view to. It's a required argument when using URL dispatch. Several optional arguments are available to specify a permission the user must have to invoke the view, and what kinds of requests the view matches (this is to limit the view to certain HTTP methods, certain routing variable values, certain query parameter values,

optional arguments:

permission

> A string naming a permission that the current user must have in order to invoke the view.

http_cache

> Affects the 'Expires' and 'Cache-Control' HTTP headers in the response. This tells the browser whether to cache the response and for how long. The value may be an integer specifying the number of seconds to cache, a `datetime.timedelta` instance, or zero to prevent caching. This is equivalent to calling `request.response.cache_expires(value)` within the view code.

This is clearly different from Pylons, and the `@action` decorator looks a bit like TurboGears. The decorator has three optional arguments:

name

> The action name, which is the target of the route. Normally this is the same as the view method name but you can override it, and you must override it when stacking multiple actions on the same view method.

renderer

> A renderer name or template filename (whose extension indicates the renderer). A renderer converts the view's return value into a Response object. Template renderers expect the view to return a dict; other renderers may allow other types. Two non-template renderers are built into Pyramid: "json" serializes the return value to JSON, and "string" calls `str()` on the return value unless it's already a Unicode object. If you don't specify a renderer, the view must return a Response object (or any object having three particular attributes described in Pyramid's Response documentation). In all cases the view can return a Response object to bypass the renderer. HTTP errors such as HTTPNotFound also bypass the renderer.

permission

> A string permission name. This is discussed in the Authorization section below.

The Pyramid developers decided to go with the return-a-dict approach because it helps in two use cases:

1. Unit testing, where you want to test the data calculated rather than parsing the HTML output. This works by default because `@action` itself does not modify the return value or arguments; it merely sets function attributes or interacts with the configurator.

2. Situations where several URLs render the same data using different templates or different renderers (like "json"). In that case, you can put multiple `@action` decorators on the same method, each with a different name and renderer argument.

Two functions in `pyramid.renderers` are occasionally useful in views:

`pyramid.renderers.`**`render`**`(`*renderer_name*, *value*, *request=None*, *package=None*`)`

> Render a template and return a string. 'renderer_name' is a template filename or renderer name. 'value' is a dict of template variables. 'request' is the request, which is needed only if the template cares about it.
>
> If the function can't find the template, try passing "zzz:templates/" as the `package` arg.

`pyramid.renderers.`**`render_to_response`**`(`*renderer_name*, *value*, *request=None*, *package=None*`)`

> Render a template, instantiate a Response, set the Response's body to the result of the rendering, and return the Response. The arguments are the same as for `render()`, except that 'request' is more important.

The handler class inherits from a base class defined in *zzz.handlers.base*:

```
"""Base classes for view handlers.
"""

class Handler(object):
    def __init__(self, request):
        self.request = request

        #c = self.request.tmpl_context
        #c.something_for_site_template = "Some value."
```

Pyramid does not require a base class but Akhet defines one for convenience. All handlers should set `self.request` in their `.__init__` method, and the base handler does this. It also provides a place to put common methods used by several handler classes, or to set `tmpl_context` (`c`) variables which are used by your site template (common to all views or several views). (You can use `c` in view methods the same way as in Pylons, although this is not recommended.)

Note that non-template renders such as "json" ignore `c` variables, so it's really only useful for HTML-only data like which stylesheet to use.

The routes are defined in *zzz/handlers/__init__.py*:

```
"""View handlers package.
"""

def includeme(config):
    """Add the application's view handlers.
    """
    config.add_handler("home", "/", "zzz.handlers.main:Main",
                       action="index")
    config.add_handler("main", "/{action}", "zzz.handlers.main:Main",
        path_info=r"/(?!favicon\.ico|robots\.txt|w3c)")
```

`includeme` is a configurator "include" function, which we've already seen. This function calls `config.add_handler` twice to create two routes. The first route connects URL "/" to the `index` view in the `Main` handler.

The second route connects all other one-segment URLs (such as "/hello" or "/help") to a same-name method in the `Main` handler. "{action}" is a path variable which will be set the corresponding substring in the URL. Pyramid will look for a method in the handler with the same action name, which can either be the method's own name or another name specified in the 'name' argument to `@action`. Of course, these other methods ("hello" and "help") don't exist in the example, so Pyramid will return 400 Not Found status.

The 'path_info' argument is a regex which excludes certain URLs from matching ("/favicon.ico", "/robots.txt", "/w3c"). These are static files or directories that would syntactically match "/{action}", but we want these to go to a later route instead (the static route). So we set a 'path_info' regex that doesn't match them.

4.5 Redirecting and HTTP errors

To issue a redirect inside a view, return an HTTPFound:

```
from pyramid.httpexceptions import HTTPFound

def myview(self):
    return HTTPFound(location=request.route_url("foo"))
    # Or to redirect to an external site
    return HTTPFound(location="http://example.com/")
```

You can return other HTTP errors the same way: `HTTPNotFound`, `HTTPGone`, `HTTPForbidden`, `HTTPUnauthorized`, `HTTPInternalServerError`, etc. These are all subclasses of both `Response` and `Exception`. Although you can raise them, Pyramid prefers that you return them instead. If you intend to raise them, you have to define an exception view that receives the exception argument and returns it, as shown in the Views chapter in the Pyramid manual. (On Python 2.4, you also have to call the instance's `.exception()` method and raise that, because you can't raise instances of new-style classes in 2.4.) A future version of Pyramid may have an exception view built-in; this would conflict with your exception view so you'd need to delete it, but there's no need to worry about that until/if it actually happens.

Pyramid catches two non-HTTP exceptions by default, `pyramid.exceptions.NotFound` and `pyramid.exceptions.Forbidden`, which it sends to the Not Found View and the Forbidden View respectively. You can override these views to display custom HTML pages.

4.5.1 More on routing and traversal

4.6 Routing methods and view decorators

Pyramid has several routing methods and view decorators. The ones we've seen, from the `pyramid_handlers` package, are:

`@action(**kw)`
: I make a method in a class into a *view* method, which `config.add_handler` can connect to a URL pattern. By definition, any class that contains view methods is a view handler. My most interesting args are 'name' and 'renderer'. If 'name' is NOT specified, the action name is the same as the method name. If 'name' IS specified, the action name can be different. If 'renderer' is specified, it indicates a renderer or template (and the template's extension indicates a renderer). If multiple `@action` decorators are put on a single method, each must have a different name, and they presumably will have different renderers too.

`config.`**`add_handler`**(*name*, *pattern*, *handler*, *action=None*, ***kw*)
: I create a route connecting the URL pattern to the handler class. If 'action' is specified, I connect the route to that specific action (a method decorated with the `@action` decorator). If 'action' is not specified, the pattern must contain a "{action}" placeholder. In that case I scan the handler class for all possible actions. It is an error to specify both "{action}" and an `action` arg. I pass extra keyword args to `config.add_route`, and keyword args in the `@action` decorator to `config.add_view`.

`config.add_handler` calls two lower-level methods which you can also call directly:

`config.`**`add_route`**(*name*, *pattern*, ***kw*)
: Create a route connecting a URL pattern directly to a view callable outside a handler. The view is specified with a 'view' arg. If the view is a function, it must take a Request argument and return a Response (or any object with the three required attributes). If it's a class, the constructor takes the Request argument and the specified method (`.__call__` by default) is called with no arguments.

`config.`**`add_view`**(***kw*)
: I register a view (specified with a 'view' arg). In URL dispatch, you normally don't call this directly but let `config.add_handler` or `config.add_route` call it for you. In traversal, you call this to register a view. The 'name' argument is the view name, which is used by traversal to choose which view to invoke.

Two others you should know about:

`config.`**`scan`**(*package=None*)
: I scan the specified package (which may be an asset spec) and import all its modules recursively, looking for functions decorated with `@view_config`. For each such function, I call `add_view` passing the decorator's args to it. I can also scan a package, in which case all submodules in the package are recursively scanned. If no package is specified, I scan the caller's package (i.e., the entire application).

I can also be called for my side effect of importing all of a package's modules even if none of them contain `@view_config`.

`@view_config(**kw)`
I decorate a function so that `config.scan` will recognize it as a view callable, and I also hold `add_view` arguments that `config.scan` will pick up and apply. I can also decorate a class or a method in a class.

4.7 View arguments

The 'name', 'renderer' and 'permission' arguments described for `@action` can also be used with `@view_config` and `config.add_view`.

`config.add_route` has counterparts to some of these such as 'view_permission'.

`config.add_view` also accepts a 'view' arg which is a view callable or asset spec. This arg is not useful for the decorators which already know the view.

The 'wrapper' arg can specify another view, which will be called when this view returns. (XXX Is this compatible with view handlers?)

4.7.1 The request object

The Request object contains all information about the current request state and application state. It's available as `self.request` in handler views, the `request` arg in view functions, and the `request` variable in templates. In pshell or unit tests you can get it via `pyramid.threadlocal.get_current_request()`. (You shouldn't use the threadlocal back door in most other cases. If something you call from the view requires it, pass it as an argument.)

Pyramid's `Request` object is a subclass of WebOb Request just like 'pylons.request' is, so it contains all the same attributes in methods like `params`, `GET`, `POST`, `headers`, `method`, `charset`, `date`, `environ`, `body`, and `body_file`. The most commonly-used attribute is `request.params`, which is the query parameters and POST variables.

Pyramid adds several attributes and methods. `context`, `matchdict`, `matched_route`, `registry`, `registry.settings`, `session`, and `tmpl_context` access the request's state data and global application data. `route_path`, `route_url`, `resource_url`, and `static_url` generate URLs, shadowing the functions in `pyramid.url`. (One function, `current_route_url`, is available only as a function.)

Rather than repeating the existing documentation for these attributes and methods, we'll just refer you to the original docs:

- Pyramd Request, Response, HTTP Exceptions, and MultiDict
- Pyramid Request API
- WebOb Request API
- Pyramid Response API
- WebOb Response API

MultiDict is not well documented so we've written our own MultiDict API page. In short, it's a dict-like object that can have multiple values for each key. `request.params`, `request.GET`, and `request.POST` are MultiDicts.

Pyramid has no pre-existing Response object when your view starts executing. To change the response status type or headers, you can either instantiate your own `pyramid.response.Response` object and return it, or use these special Request attributes defined by Pyramid:

```
request.response_status = "200 OK"
request.response_content_type = "text/html"
request.response_charset = "utf-8"
request.response_headerlist = [
    ('Set-Cookie', 'abc=123'), ('X-My-Header', 'foo')]
request.response_cache_for = 3600    # Seconds
```

Akhet adds one Request attribute. `request.url_generator`, which is used to implement the `url` template global described below.

4.7.2 Templates

Pyramid has built-in support for Mako and Chameleon templates. Chameleon runs only on CPython and Google App Engine, not on Jython or other platforms. Jinja2 support is available via the `pyramid_jinja2` package on PyPI, and a Genshi emulator using Chameleon is in the `pyramid_chameleon_genshi` package.

Whenever a renderer invokes a template, the template namespace includes all the variables in the view's return dict, plus the following:

request
: The current request.

context
: The context (same as `request.context`).

renderer_name
: The fully-qualified renderer name; e.g., "zzz:templates/foo.mako".

renderer_info
: An object with attributes `name`, `package`, and `type`.

The subscriber in your application adds the following additional variables:

c, tmpl_context
: `request.tmpl_context`

h
: The helpers module, defined as "zzz.helpers". This is set by a subscriber callback in your application; it is not built into Pyramid.

session
: `request.session`.

url
: In Akhet, a URLGenerator object. In Pyramid's built-in application templates that use URL dispatch, an alias to the `route_url` *function*, which requires you to pass the route name as the first arg and the request as the second arg.

The URLGenerator object has convenience methods for generating URLs based on your application's routes. See the complete list on the API page.

By default the generator creates unqualified URLs (i.e., without the "scheme://hostname" prefix) if the underlying Pyramid functions allow it. To get absolute URLs throughout the application, edit *zzz/subscribers.py*, go to the line where the URLGenerator is instantiated, and change the 'qualified' argument to True. Pylons traditionally uses unqualified URLs, while Pyramid traditionally uses qualified URLs. Note that qualified URLs may be wrong if the application is running behind a reverse proxy! (E.g., Apache's mod_proxy.) The generated URL may be "http://localhost:5000" which is correct for the application but invalid to the end user (who needs the proxy's URL, "https://example.com").

4.8 Advanced template usage

If you need to fill a template within view code or elsewhere, do this:

```
from pyramid.renderers import render
variables = {"foo": "bar"}
html = render("mytemplate.mako", variables, request=request)
```

There's also a `render_to_response` function which invokes the template and returns a Response, but usually it's easier to let `@action` or `@view_config` do this. However, if your view has an if-stanza that needs to override the template specified in the decorator, `render_to_response` is the way to do it.

```
@action(renderer="index.html")
def index(self):
    records = models.MyModel.all()
    if not records:
        return render_to_response("no_records.html")
    return {"records": records}
```

For further information on templating see the Templates section in the Pyramid manual, the Mako manual, and the Chameleon manual. You can customize Mako's TemplateLookup by setting "mako.*" variables in the INI file.

4.9 Site template

Most applications using Mako will define a site template something like this:

```
<!DOCTYPE html>
<html>
  <head>
    <title>${self.title()}</title>
    <link rel="stylesheet" href="${application_url}/default.css"
        type="text/css" />
  </head>
  <body>

<!-- *** BEGIN page content *** -->
${self.body()}
<!-- *** END page content *** -->
  </body>
</html>
<%def name="title()" />
```

Then the page templates can inherit it like so:

```
<%inherit file="/site.html" />
<%def name="title()">My Title</def>
... rest of page content goes here ...
```

4.9.1 Static files

Pyramid has five ways to serve static files. Each way has different advantages and limitations, and requires a different way to generate static URLs.

```
config.add_static_route
```

This is the Akhet default, and is closest to Pylons. It serves the static directory as an overlay on "/", so that URL "/robots.txt" serves "zzz/static/robots.txt", and URL "/images/logo.png" serves "zzz/static/images/logo.png". If the file does not exist, the route will not match the URL and Pyramid will try the next route or traversal. You cannot use any of the URL generation methods with this; instead you can put a literal URL like "${application_url}/images/logo.png" in your template.

Usage:

```
config.include('akhet')
config.add_static_route('zzz', 'static', cache_max_age=3600)
# Arg 1 is the Python package containing the static files.
# Arg 2 is the subdirectory in the package containing the files.
```

config.add_static_view

This is Pyramid's default algorithm. It mounts a static directory under a URL prefix such as "/static". It is not an overlay; it takes over the URL prefix completely. So URL "/static/images/logo.png" serves file "zzz/static/images/logo.png". You cannot serve top-level static files like "/robots.txt" and "/favicon.ico" using this method; you'll have to serve them another way.

Usage:

```
config.add_static_view("static", "zzz:static")
# Arg 1 is the view name which is also the URL prefix.
# It can also be the URL of an external static webserver.
# Arg 2 is an asset spec referring to the static directory/
```

To generate "/static/images/logo.png" in a Mako template, which will serve "zzz/static/images/logo.png":

```
href="${request.static_url('zzz:static/images/logo.png')}
```

One advantage of add_static_view is that you can copy the static directory to an external static webserver in production, and static_url will automatically generate the external URL:

```
# In INI file
static_assets = "static"
# -OR-
static_assets = "http://staticserver.com/"

config.add_static_view(settings["static_assets"], "zzz:static")

href="${request.static_url('zzz:static/images/logo.png')}"
## Generates URL "http://staticserver.com/static/images/logo.png"
```

Other ways

There are three other ways to serve static files. One is to write a custom view callable to serve the file; an example is in the Static Assets section of the Pyramid manual. Another is to use `paste.fileapp.FileApp` or `paste.fileapp.DirectoryApp` in a view. (More recent versions are in the "PasteOb" distribution.) These three ways can be used with `request.route_url()` because the route is an ordinary route. The advantage of these three ways is that they can serve a static file or directory from a normal view callable, and the view can be protected separately using the usual authorization mechanism.

4.9.2 Session, flash messages, and secure forms

Pyramid's session object is `request.session`. It has its own interface but uses Beaker on the back end, and is configured in the INI file the same way as Pylons' session. It's a dict-like object and can store any pickleable value.

It's pulled from persistent storage only if it's accessed during the request processing, and it's (re)saved only if the data changes.

Unlike Pylons' sesions, you don't have to call `session.save()` after adding or replacing keys because Pyramid does that for you. But you do have to call `session.changed()` if you modify a mutable value in place (e.g., a session value that's a list or dict) because Pyramid can't tell that child objects have been modified.

You can call `session.invalidate()` to discard the session data at the end of the request. `session.created` is an integer timestamp in Unix ticks telling when the session was created, and `session.new` is true if it was created during this request (as opposed to being loaded from persistent storage).

Pyramid sessions have two extra features: flash messages and a secure form token. These replace `webhelpers.pylonslib.flash` and `webhelpers.pylonslib.secure_form`, which are incompatible with Pyramid.

Flash messages are a session-based queue. You can push a message to be displayed on the next request, such as before redirecting. This is often used after form submissions, to push a success or failure message before redirecting to the record's main screen. (This is different from form validation, which normally redisplays the form with error messages if the data is rejected.)

To push a message, call `request.session.flash("My message.")` The message is normally text but it can be any object. Your site template will then have to call `request.session.pop_flash()` to retrieve the list of messages, and display then as it wishes, perhaps in <div>'s or a <ul>. The queue is automatically cleared when the messages are popped, to ensure they are displayed only once.

The full signature for the flash method is:

```
session.flash(message, queue='', allow_duplicate=True)
```

You can have as many message queues as you wish, each with a different string name. You can use this to display warnings differently from errors, or to show different kinds of messages at different places on the page. If `allow_duplicate` is false, the message will not be inserted if an identical message already exists in that queue. The `session.pop_flash` method also takes a queue argument to specify a queue. You can also use `session.peek_flash` to look at the messages without deleting them from the queue.

The secure form token prevents cross-site request forgery (CSRF) attacts. Call `session.get_csrf_token()` to get the session's token, which is a random string. (The first time it's called, it will create a new random token and store it in the session. Thereafter it will return the same token.) Put the token in a hidden form field. When the form submission comes back in the next request, call `session.get_csrf_token()` again and compare it to the hidden field's value; they should be the same. If the form data is missing the field or the value is different, reject the request, perhaps by returning a forbidden status. `session.new_csrf_token()` always returns a new token, overwriting the previous one if it exists.

4.9.3 WebHelpers and forms

Most of WebHelpers works with Pyramid, including the popular `webhelpers.html` subpackage, `webhelpers.text`, and `webhelpers.number`. Pyramid does not depend on WebHelpers so you'll have to add the dependency to your application if you want to use it. The only part that doesn't work with Pyramid is the `webhelpers.pylonslib` subpackage, which depends on Pylons' special globals.

We are working on a form demo that compares various form libraries: Deform, Formish, FormEncode/htmlfill.

To organize the form display-validate-action route, we recommend the `pyramid_simpleform` package. It replaces `@validate` in Pylons. It's not a decorator because too many people found the decorator too inflexible: they ended up copying part of the code into their action method.

WebHelpers 1.3 has some new URL generator classes to make it easier to use with Pyramid. See the `webhelpers.paginate` documentation for details. (Note: this is *not* the same as Akhet's URL generator; it's a different kind of class specifically for the paginator's needs.)

4.9.4 Shell

paster pshell is similar to Pylons' "paster shell". It gives you an interactive shell in the application's namespace with a dummy request. Unlike Pylons, you have to specify the application section on the command line because it's not "main". Akhet, for convenience, names the section "myapp" regardless of the actual application name.

```
$ paster pshell development.ini myapp
Python 2.6.6 (r266:84292, Sep 15 2010, 15:52:39)
[GCC 4.4.5] on linux2
Type "help" for more information. "root" is the Pyramid app root object, "registry" is the Pyramid re
>>> registry.settings["sqlalchemy.url"]
'sqlite:////home/sluggo/exp/pyramid-docs/main/workspace/Zzz/db.sqlite'
>>> import pyramid.threadlocal
>>> request = pyramid.threadlocal.get_current_request()
>>>
```

As the example above shows, the interactice namespace contains two objects initially: `root` which is the root object, and `registry` from which you can access the settings. To get the request, you have to use Pyramid's threadlocal library to fetch it. This is one of the few places where it's recommended to use the threadlocal library.

4.9.5 Deployment

Deployment is the same for Pyramid as for Pylons. Use "paster serve" with mod_proxy, or mod_wsgi, or whatever else you prefer.

CHAPTER

FIVE

DEFAULT TEMPLATES AND STYLESHEET

The default home page was redesigned in Akhet 1.0 final to be a simple base you can start with and add to if you wish. It consists of four files:

- A home page, *zzz/templates/index.html*
- A site template, *zzz/templates/site.html*
- A stylesheet, *zzz/static/stylesheets/default.css*
- A "reset" stylesheet, *zzz/static/stylesheets/reset.css*

The HTML files are Mako templates. The stylesheets are static files.

5.1 index.html

This is a page template, so it contains the specific text for this page. It contains just the HTML body, not the tags around it or the HTML header. Those will be added by the site template. The first three lines are Mako constructs:

```
<%inherit file="/site.html" />
<%def name="title()">${project}</%def>
<%def name="body_title()">Hello, ${project}!</%def>
```

Line 1 makes the template inherit from the site template, which will add the surrounding HTML tags.

Lines 2 and 3 are Mako methods; they return values which will be used by the site template. Line 2 is the title for the "<title>" tag. Line 3 is the title to display inside the page. 'project' is a variable the view method passes via its return dict. The rest of the page is ordinary HTML so we won't bother showing it.

5.2 Site template

The site template contains everything around the page content: the "<html>" container tag, the HTML header, and the parts of the page body that are the same on every page. The most important construct here is the "${self.body()}" placeholder, which is where the entire page template will be rendered. Mako's 'self' construct chooses the highest-level variable available, which allows a page template to override a default value in a parent template the way Python class attributes override superclass attributes.

The "<head>" section contains the usual title, character set, stylesheet, and the like. You can modify these as you wish.

The "<body>" section contains a standardized header and footer; you can modify these as you wish to put the same doodads on all your pages.

Three "<%def>" methods are defined at the bottom of the file, which page templates can override:

head_extra()
: Override this to put extra tags into the <head> section like page-specific styles, Javascript, or metadata. The default is empty.

title()
: We saw this in the page template. Put the title for the <title> tag here. The default is empty: no title.

body_title()
: Put the title for the page body here. The default is to be the same as `title`. You can override it if you want different wording, or to put embedded HTML tags in the body title. (The <title> can't have embedded HTML tags: the browser would display them literally.)

The site template also has a stanza to display flash messages:

```
<div id="content">
<div id="flash-messages">
% for message in request.session.pop_flash():
    <div class="info">${message}</div>
% endfor
</div>
```

Flash messages are a queue of messages in the session which are displayed on the next page rendered. Normally a view will push a success or failure message and redirect, and the redirected-to page will display the message. If you call 'pop_flash' without a queue name, the default queue is used. This is enough for many programs. You can also define multiple queues for different kinds of messages, and then pop each queue separately and display it in a different way. For instance:

```
% for message in request.session.pop_flash("error"):
    <div class="error">${message}</div>
% endfor
% for message in request.session.pop_flash("warn"):
    <div class="error">${warning}</div>
% endfor
```

5.3 Reset stylesheet

This is an industry-standard reset stylesheet by Eric Meyer, which is in the public domain. The original site is http://meyerweb.com/eric/tools/css/reset/ . It resets all the tag styles to be consistent across browsers.

The top part of the page is Meyer's original stylesheet; the bottom contains some overrides. Meyers does remove some attributes which have generally been assumed to be intrinsic to the tag, such as margins around <p> and <h*>. His reasoning is that you should start with nothing and consciously re-add the styles you want. Some people may find this attitude to be overkill. The reset stylesheet is just provided as a service if you want to use it. In any case, we re-add some expected styles, and I also set <dt> to bold which is a pet peeve of mine.

If you want something with more bells and whistles, some Pyramid developers recommend HTML5 Boilerplate. It's also based on Meyer's stylesheet.

5.4 Default stylesheet

This is the stylesheet referenced in the page template; it inherits the reset stylesheet. It defines some styles the default home page needs. You'll probably want to adjust them for your layout.

The bottom section has styles for flash messages. The ".info" stanza is used by the default application. The ".warning" and ".error" styles are not used by default but are provided as extras.

CHAPTER

SIX

TRANSACTION MANAGER

Akhet's SQALchemy configuration includes the pyramid_tm transaction manager. This is a feature which TurboGears has long had but Pylons has not. The transaction manager provides an automatic commit or rollback at the end of the request processing, depending on whether an error has occurred.

(TurboGears uses a different transaction manager "repoze.tm2" which does essentially the same thing but requires a middleware. pyramid_tm does not have a middleware.)

6.1 How it works

After the view returns a response, a subscriber callback will automatically commit all database changes in the Session unless an uncaught exception has occurred, in which case it will roll back the changes. It will also roll back the changes if the HTTP status is 4xx or 5xx. Finally, it clears the Session for the next request.

You can still commit and roll back explicitly in your view, but you'll have to use the `transaction` module instead of calling the Session methods directly:

```
import transaction
transaction.commit()
# Or:
transaction.abort()
```

You may want to do this if you want to commit a lot of data a little bit at a time.

You can also poison the transaction to prevent *any* database writes during this request, including those performed by other parts of the application or middleware. To do this, call:

```
transaction.doom()
```

Of course, this doesn't affect changes that have already been committed.

The implementation is a combination of three packages that work together. `transaction` is a generic transaction manager. `zope.sqlalchemy` applies this to SQLAlchemy by exposing a `ZopeTransactionExtension`, which is a SQLAlchemy session extension (a class that enhances the session's behavior). `pyramid_tm` takes care of issuing the commit or rollback at the end of the request processing.

SQLAHelper maintains a ZopeTransactionExtension in the `sqlahelper._zte` variable. It automatically configures the Session to use that extension.

You can customize the circumstances under which an automatic rollback occurs by defining a "commit veto" function. This is described in the pyramid_tm documentation.

6.2 Disabling the transaction manager

If you don't want managed transactions:

1. Delete the `config.include("pyramid_tm")` line in the `main` function.
2. Reconfigure the Session to not use the transaction extension:

   ```
   sqlahelper.get_session().config(extension=None)
   ```

If you disable the manager, you'll have to call `Session.commit()` or `Session.rollback()` yourself in your views. You'll also have to configure the application to remove the session at the end of the request. This would be in an event subscriber but I'm not sure which one.

6.3 Caveat: adding your own session extensions

If you modify the `extension` session option in any way you'll lose the transaction extension unless you re-add it. The extension lives in the semi-private `_zte` variable in the library. Here's the proper way to add your own extension while keeping the transaction extension:

```
Session = sqlahelper.get_session()
Session.configure(extension=[MyWonderfulExtension(), sqlahelper._zte])
```

6.4 Bypassing the transaction manager without disabling it

In special circumstances you may want to do a particular database write while allowing the transaction manager to roll back all other writes. For instance, if you have a separate access log database and you want to log all responses, even failures. In that case you can create a second SQLAlchemy session using `sqlalchemy.orm.sessionmaker` – one that does *not* use the transaction extension – and use that session with that engine to insert and commit the log record.

CHAPTER

SEVEN

MODEL EXAMPLES

This chapter gives some examples for writing your application models. These are based on the SQLAlchemy documentation, which should be your primary guide.

These models assume SQLAlchemy 0.6.x and Python >= 2.6. For earlier versions of Python, use the `%` operator instead of the the string `.format` method.

7.1 A simple one-table model

```
import sqlahelper
import sqlalchemy as sa
import sqlalchemy.orm as orm

Base = sqlahelper.get_base()
Session = sqlahelper.get_session()

class User(Base):
    __tablename__ = "users"

    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(100), nullable=False)
    email = sa.Column(sa.Unicode(100), nullable=False)
```

This model has one ORM class, `User` corresponding to a database table `users`. The table has three columns: `id`, `name`, and `user`.

7.2 A three-table model

We can expand the above into a three-table model suitable for a medium-sized application.

```
import sqlahelper
import sqlalchemy as sa
import sqlalchemy.orm as orm

Base = sqlahelper.get_base()
Session = sqlahelper.get_session()

class User(Base):
    __tablename__ = "users"
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(100), nullable=False)
    email = sa.Column(sa.Unicode(100), nullable=False)

    addresses = orm.relationship("Address", order_by="Address.id")
    activities = orm.relationship("Activity",
        secondary="assoc_users_activities")

    @classmethod
    def by_name(class_):
        """Return a query of users sorted by name."""
        User = class_
        q = Session.query(User)
        q = q.order_by(User.name)
        return q


class Address(Base):
    __tablename__ = "addresses"

    id = sa.Column(sa.Integer, primary_key=True)
    user_id = foreign_key_column(None, sa.Integer, "users.id")
    street = sa.Column(sa.Unicode(40), nullable=False)
    city = sa.Column(sa.Unicode(40), nullable=False)
    state = sa.Column(sa.Unicode(2), nullable=False)
    zip = sa.Column(sa.Unicode(10), nullable=False)
    country = sa.Column(sa.Unicode(40), nullable=False)
    foreign_extra = sa.Column(sa.Unicode(100, nullable=False))

    def __str__(self):
        """Return the address as a string formatted for a mailing label."""
        state_zip = u"{0} {1}".format(self.state, self.zip).strip()
        cityline = filterjoin(u", ", self.city, state_zip)
        lines = [self.street, cityline, self.foreign_extra, self.country]
        return filterjoin(u"|n", *lines) + u"\n"


class Activity(Base):
    __tablename__ = "activities"

    id = sa.Column(sa.Integer, primary_key=True)
    activity = sa.Column(sa.Unicode(100), nullable=False)


assoc_users_activities = sa.Table("assoc_users_activities", Base.metadata,
    foreign_key_column("user_id", sa.Integer, "users.id"),
    foreign_key_column("activities_id", sa.Unicode(100), "activities.id"))

# Utility functions
def filterjoin(sep, *items):
    """Join the items into a string, dropping any that are empty.
    """
    items = filter(None, items)
    return sep.join(items)

def foreign_key_column(name, type_, target, nullable=False):
    """Construct a foreign key column for a table.
    ``name`` is the column name. Pass ``None`` to omit this arg in the
    ``Column`` call; i.e., in Declarative classes.

    ``type_`` is the column type.

    ``target`` is the other column this column references.

    ``nullable``: pass True to allow null values. The default is False
    (the opposite of SQLAlchemy's default, but useful for foreign keys).
    """
    fk = sa.ForeignKey(target)
    if name:
        return sa.Column(name, type_, fk, nullable=nullable)
    else:
        return sa.Column(type_, fk, nullable=nullable)
```

This model has a `User` class corresponding to a `users` table, an `Address` class with an `addresses` table, and an `Activity` class with `activities` table. `users` is in a 1:Many relationship with `addresses`. `users` is also in a Many:Many`` relationship with `activities` using the association table `assoc_users_activities`. This is the SQLAlchemy "declarative" syntax, which defines the tables in terms of ORM classes subclassed from a declarative `Base` class. Association tables do not have an ORM class in SQLAlchemy, so we define it using the `Table` constructor as if we weren't using declarative, but it's still tied to the Base's "metadata".

We can add instance methods to the ORM classes and they will be valid for one database record, as with the `Address.__str__` method. We can also define class methods that operate on several records or return a query object, as with the `User.by_name` method.

There's a bit of disagreement on whether `User.by_name` works better as a class method or static method. Normally with class methods, the first argument is called `class_` or `cls` or `klass` and you use it that way throughout the method, but in ORM queries it's more normal to refer to the ORM class by its proper name. But if you do that you're not using the `class_` variable so why not make it a static method? But the method does belong to the class in a way that an ordinary static method does not. I go back and forth on this, and sometimes assign `User = class_` at the beginning of the method. But none of these ways feels completely satisfactory, so I'm not sure which is best.

7.3 Common base class

You can define a superclass for all your ORM classes, with common class methods that all of them can use. You can't use SQLAHelper's declarative base in this case because it's already defined with another superclass, so you'll have to define your own declarative base:

```
class ORMClass(object):
    @classmethod
    def query(class_):
        return Session.query(class_)

    @classmethod
    def get(class_, id):
        return Session.query(class_).get(id)

Base = declarative.declarative_base(cls=ORMClass)

class User(Base):
    __tablename__ = "users"

    # Column definitions omitted
```

Then you can do things like this in your views:

```
user_1 = models.User.get(1)
q = models.User.query()
```

Whether this is a good thing or not depends on your perspective.

7.4 Multiple databases

The default configuration in the main function configures one database. To connect to multiple databases, list them all in *development.ini* under distinct prefixes. You can put additional engine arguments under the same prefixes. For instance:

Then modify the main function to add each engine. You can also pass even more engine arguments that override any same-name ones in the INI file.

```
engine = sa.engine_from_config(settings, prefix="sqlalchemy.",
    pool_recycle=3600, convert_unicode=True)
stats = sa.engine_from_config(settings, prefix="stats.")
sqlahelper.add_engine(engine)
sqlahelper.add_engine(stats, "stats")
```

In this scenario, the 'engine' engine was added without a name (no second argument), so it becomes the default engine named "default". The contextual session is bound to it, and the declarative base's metadata is bound to it too. To retrieve it later, call `sqlahelper.get_engine()`.

The 'stats' engine was added under the name "stats", so it is not bound to anything. To use it, you must pass the 'bind=stats' argument to any ORM method or SQL method that executes a query or command, or execute the code directly on the engine itself:

```
# ORM example
records = Session.query(MyTable).all(bind=stats)

# SQL example
sql = sa.select([MyTable.__table__])
rslt = stats.execute(sql)
records = rslt.fetchall()
```

If you're in a function and need a reference to the engine, retrieve it by name: `sqlahelper.get_engine("stats")`.

You can, of course, create an engine directly without going through the application settings:

```
engine = sa.create_engine("mysql://me:PASSWORD@localhost/farm")
sqlahelper.add_engine(engine, "farm")
```

7.5 Two engines, but no default engine

In this scenario, two engines are equally important, and neither is predominent enough to deserve being the default engine. This is useful in applications whose main job is to copy data from one database to another. The configuration is the same except that we name both engines:

```
sqlahelper.add_engine(db1, "db1")
sqlahelper.add_engine(db2, "db2")
```

Because there is no default engine, you will have to use the 'bind' argument for all queries, or execute them directly on the engine.

7.6 Different tables bound to different engines

You can bind different ORM classes to different engines in the same database session. Configure your application with no default engine, and then call the Session's `.configure` method with the `binds=` argument to specify which classes go to which engines. For instance:

```
sqlahelper.add_engine(db1, "db1")
sqlahelper.add_engine(db2, "db2")
Session = sqlahelper.get_session()
import zzz.models as models
binds = {models.Person: db1, models.Score: db1}
Session.configure(binds=binds)
```

The keys in the `binds` dict can be SQLAlchemy ORM classes, table objects, or mapper objects.

7.7 Reflected tables

Reflected tables pose a dilemma because they depend on a live database connection in order to be initialized. But the engine may not be configured yet when the model is imported. SQLAHelper does not address this issue directly. Pylons 1 models traditionally have a `model.init_model(engine)` function which performs any initialization that requires a live connection. Pyramid applications typically do not need this function because the Session, engines, and base are initialized in the `sqlahelper` library before the model is imported. But in the case of reflection, you'll probably need an `init_model` function that sets global variables. You'll just have to remember to call the function before using anything in the model.

If you're using SQLAlchemy's declarative syntax as in the examples above, we *think* you'd have to define the entire ORM class inside the function, and use a `global` statement to put the class into the module namespace.

If you're not using declarative, the ORM class can be defined at module level, but the table will have to be defined in the function with a `global` statement, and the mapper call will also have to be in the function.

CHAPTER

EIGHT

AUTHENTICATION AND AUTHORIZATION

This chapter is contributed by Eric Rasmussen.

Pyramid has built-in authentication and authorization capibalities that make it easy to restrict handler actions. Here is an overview of the steps you'll generally need to take:

1. Create a root factory in your model that associates allow/deny directives with groups and permissions
2. Create users and groups in your model
3. Create a callback function to retrieve a list of groups a user is subscribed to based on their user ID
4. Make a "forbidden view" that will be invoked when a Forbidden exception is raised.
5. Create a login action that will check the username/password and remember the user if successful
6. Restrict access to handler actions by passing in a permission='somepermission' argument to the `@action` decorator
7. Wire it all together in your config

You can get started by adding an import statement and custom root factory to your model:

```
from pyramid.security import Allow, Everyone

class RootFactory(object):
    __acl__ = [ (Allow, Everyone, 'everybody'),
                (Allow, 'basic', 'entry'),
                (Allow, 'secured', ('entry', 'topsecret'))
              ]
    def __init__(self, request):
        pass
```

The custom root factory generates objects that will be used as the context of requests sent to your web application. The first attribute of the root factory is the ACL, or access control list. It's a list of tuples that contain a directive to handle the request (such as Allow or Deny), the group that is granted or denied access to the resource, and a permission (or optionally a tuple of permissions) to be associated with that group.

The example access control list above indicates that we will allow everyone to view pages with the 'everybody' permission, members of the basic group to view pages restricted with the 'entry' permission, and members of the secured group to view pages restricted with either the 'entry' or 'topsecret' permissions. The special principal 'Everyone' is a built-in feature that allows any person visiting your site (known as a principal) access to a given resource.

For a user to login, you can create a handler that validates the login and password (or any additional criteria) submitted through a form. You'll typically want to add the following imports:

```
from pyramid.httpexceptions import HTTPFound
from pyramid.security import remember, forget
```

Once you validate a user's login and password against the model, you can set the headers to "remember" the user's ID, and then you can redirect the user to the home page or url they were trying to access:

```
# retrieve the userid from the model on valid login
headers = remember(self.request, userid)
return HTTPFound(location=someurl, headers=headers)
```

Note that in the call to the remember function, we're passing in the user ID we retrieved from the database and stored in the variable 'userid' (an arbitrary name used here as an example). However, you could just as easily pass in a username or other unique identifier. Whatever you decide to "remember" is what will be passed to the groupfinder callback function that returns a list of groups a user belongs to. If you import `authenticated_userid`, which is a useful way to retrieve user information in a handler action, it will return the information you set the headers to "remember".

To log a user out, you "forget" them, and use HTTPFound to redirect to another url:

```
headers = forget(self.request)
return HTTPFound(location=someurl, headers=headers)
```

Before you restrict a handler action with a permission, you will need a callback function to return a list of groups that a user ID belongs to. Here is one way to implement it in your model, in this case assuming you have a Groups object with a groupname attribute and a Users object with a mygroups relation to Groups:

```
def groupfinder(userid, request):
    user = Users.by_id(userid)
    return [g.groupname for g in user.mygroups]
```

As an example, you could now import and use the @action decorator to restrict by permission, and authenticated_userid to retrieve the user's ID from the request:

```
from pyramid_handlers import action
from pyramid.security import authenticated_userid
from models import Users

class MainHandler(object):
    def __init__(self, request):
        self.request = request

    @action(renderer='welcome.html', permission='entry')
    def index(self):
        userid = authenticated_userid(self.request)
        user = Users.by_id(userid)
        username = user.username
        return {'currentuser':username}
```

This gives us a very simple way to restrict handler actions and also obtain information about the user. This example assumes we have a Users class with a convenience class method called by_id to return the user object. You can then access any of the object's attributes defined in your model (such as username, email address, etc.), and pass those to a template as dictionary key/values in your return statement.

If you would like a specific handler action to be called when a forbidden exception is raised, you need to add a forbidden view. You can't use `@action` for this, so you'll either have to use `@view_config` and enable scanning in your main function, or define it as a view function and register it with `config.add_view`. Here's the first way:

```
# In a view handler
@view_config(renderer='myapp:templates/forbidden.html',
             context='pyramid.exceptions.Forbidden')
@action(renderer='forbidden.html')
def forbidden(self):
    # actions to execute

# In the main function.
config.scan("myapp.handlers")
```

And here's the second:

```
# In myapp/handlers/__init__.py or myapp/views.py
def forbidden(request):
    # actions to execute

# In the main function or in some include function
config.add_view("myapp.views.forbidden",
    renderer="myapp:templates/forbidden.html",
    context="pyramid.exceptions.Forbidden")
```

The last step is to configure __init__.py to use your auth policy. Make sure to add these imports:

```
from pyramid.authentication import AuthTktAuthenticationPolicy
from pyramid.authorization import ACLAuthorizationPolicy
from models import groupfinder
```

In your main function you'll want to define your auth policies so you can include them in the call to Configurator:

```
authn_policy = AuthTktAuthenticationPolicy('secretstring',
                                           callback=groupfinder)
authz_policy = ACLAuthorizationPolicy()
config = Configurator(settings=settings,
                      root_factory='myapp.models.RootFactory',
                      authentication_policy=authn_policy,
                      authorization_policy=authz_policy)
config.scan()
```

You only need to add `config.scan()` if you are using the `@view_config` decorator.

The capabilities for authentication and authorization in Pyramid are very easy to get started with compared to using Pylons and repoze.what. The advantage is easier to maintain code and built-in methods to handle common tasks like remembering or forgetting users, setting permissions, and easily modifying the groupfinder callback to work with your model. For cases where it's manageable to set permissions in advance in your root factory and restrict individual handler actions, this is by far the simplest way to get up and running while still offering robust user and group management capabilities through your model.

However, if your application requires the ability to create/edit/delete permissions (not just access through group membership), or you require the use of advanced predicates, you can either build your own auth system (see the Pyramid docs for details) or integrate an existing system like repoze.what.

CHAPTER

NINE

TESTING

XXX Need somebody to write this chapter. Pyramid makes it easier to write unit tests than Pylons.

CHAPTER

TEN

INTERNATIONALIZATION

XXX Need somebody to write this chapter.

CHAPTER

ELEVEN

MIGRATING A PYLONS APPLICATION TO AKHET

A user wrote to the "pylons-discuss" mailing list:

> I've been developing a Pylons application over the past year or so. The application has over 20 controllers with more than six functions each and makes use of:
>
> - Pylons>=1.0
> - SQLAlchemy>=0.5,<=0.5.99
> - Mako
> - WebHelpers>=1.0
> - FormBuild>=3.0,<=3.99
> - AuthKit>=0.4.3,<=0.4.99
>
> Having read your migration guide I find it hard to understand what the most effective way of porting my application to Pyramid would be. Should I start changing things in my exisitng project? What changes do I need to make? Or would it be more productive to start a project from fresh using the paster pyramid_sqla template and copy and paste functionality from my Pylons project?

The migration guide has become this manual, and pyramdid_sqla has become Akhet, but the question is still relevant.

If this were my application, I'd make only one change to the existing project: bring it up to SQLAlchemy 0.6. (Or 0.7 if it's out by the time you read this.) That will be easiest to do in your familiar environment where it'll be easier to debug anything that might go wrong. One issue when upgrading from 0.5: 0.6 issues a warning if you assign a non-Unicode string to a Unicode column (or if you have 'convert_unicode=True' which treats all String columns into Unicode columns). You'll either have to convert all your string literals or disable the warning if it becomes annoying. On the other hand, 0.6 also leverages the native Unicode support that exists in some database drivers; this makes it more efficient on SQLite and PostgreSQL/`psycopg2` at least.

Then I'd make a new project using the "akhet" skeleton, and start porting your code to it. The differences between Pylons and Pyramid are so large that it's not worth trying to upgrade the application in place. (Note: in this chapter I don't distinguish between Akhet features and generic Pyramid features; that's what the rest of this manual is about.)

After creating the application, delete everything in the static directory and replace it with your files. Then copy your helpers to *zzz/lib/helpers.py*, add any necessary imports to *setup.py*, and install the application. ("python setup.py develop", "pip install -e .", or "pip install .") Installing the application updates the *ZZZ.egg-info* metadata and installs the new dependencies.

Then empty out *index.html* and replace it with some minimal content. If you're using a site template, copy it now and make index.html inherit from it. There's no routing yet so you'll have to use hardcoded URLs or comment out the `${url(...)}` calls. You've already migrated the helpers so any `${h.foo()}` calls should work. If the site template depends on `c` variables set in the base controller, put equivalent attributes in `zzz.handlers.base.Base.__init__` under `self.request.tmpl_context`. (This object is available in template as both `tmpl_context` and `c`, as in Pylons.) Then run the application and make sure the home page works.

Then I'd work on the model, which probably has few if any dependencies on the rest of the application. You should be able to copy your tables and classes unchanged, and make them use the `Session` retrieved from SQLAHelper. You won't need `init_model` unless you're using reflection; if you are, add a call to it in the main function.

Copy your existing database, adjust *development.ini* for it, and empty out the *index.html* template. Change the index view method to perform a simple model lookup, pass the record to the template in the view's return dict, and make the template display it. Run the application to make sure it works.

Now you can start porting your controllers to view handlers one at a time, copying the templates as you need them. The templates will remain the same except ${c.foo} changes to ${foo} – and make sure that doesn't overlap with a local template variable (a 'for' variable, 'def' argument, '<% %>' variable, etc).

In the view method, path variable arguments change to `self.request.matchdict['var']`. GET/POST variables remain the same: `self.request.params` (or `self.request.GET` and `self.request.POST`). Instead of setting `c` variables, return a dict of template variables, and set the `@action` decorator with the 'renderer' arg pointing to the template.

You *can* set `c` variables in the view for the template, assigning them to `self.request.tmpl_context`, but this is unusual in Pyramid views. It's still useful, however, to share data between the base class's .__init__ method, the template, the view, and utility methods the view calls).

Any state information you need (which in Pylons would be in the special globals) is under `self.request` somewhere.

If you have your own `app_globals` attribtues, migrate these to the `settings` dict if feasable. You can set these in the application's main() function, and retrieve them as `self.request.registry.settings` I think it's called.

As in Pylons, the view handler is instantiated for each request, so any `self` attributes you set will be visible only in the current request.

When you've finished your handler, you'll have to make a route to it. (Actually, I create applications by first writing down the URL structure, then defining my routes and then my views. But in that case you'd have to comment out routes pointing to views that don't exist yet.) In the default Akhet application, routes are defined in an `includeme` function in *zzz/handlers/__init__.py*, which is included in the main function via a `config.include()` call.

As in Pylons, you can make a separate route for each view (with 'action' as a keyword argument), or a general route for the entire handler (with "{action}" as a path variable). You can't make a single route to multiple handlers; so there's no equivalent to Pylons' "/{controller}/{action}" route.

Once the routes are defined, generating URLs is similar to Pylons. Call `${url("route_name", arg1="value1")}`. To generate a URL based on the current request's route, call `${url.current(arg1="value1")}`. Unlike Pylons, these methods do not convert keyword args not corresponding to path variables into query parameters. Instead, pass a dict of query params via the '_query' argument. (To propagate the current request's params, '_query=request.GET'.) You can also pass '_anchor', which will be the page's "#anchor" value. '_app_url' replaces the "scheme://host" prefix *if the URL generator was configured to produce absolute URLs*. (It's not by default; see the Templates section and *API <api.html>* page for details.)

If you want to create a link to the current URL with different query parameters, see `webhelpers.util.update_params` in the WebHelpers package. The current URL is `${request.path}` (path only without query), `${request.path_url}` (absolute without query), and `${request.url}` (absolute with query).

Making URLs to static files is trickier because the methods above won't work for that. `${url.app}` is the application's base URL, so you can do `${url.app}subdir/filename.css` to semi-hardcode the URL.

Or you can switch to Pyramid's default way of serving static files (add_static_view), which exposes the "/static" prefix to the user. Then you can use `${request.static_url()}`, and even set it up to generate external URLs to a static media server. But this won't work for top-level URLs ("/favicon.ico"). URLGenerator does not have a shadow method for static_url; there's a commented method in the source but we're not sure of the best API. You can define a method in a subclass and use it in *subscribers.py*. There's also a commented method to serve static files from the Deform library if you're using that.

Speaking of forms, you can try sticking with FormBuild or switch to Deform or one of the other libraries. I don't know whether FormBuild is compatible with Pyramid, and I'm not sure how well maintained it is. Deform was written by chrism who wrote Pyramid, so it has good support. The Pylons form standard (FormEncode + WebHelpers) also works under Pyramid. The `@validate` decorator does not exist in Pyramid; it was seen as flawed and not worth porting. Most Akhet users use "pyramid_simpleform", an add-on package in PyPI that provides a way to organize form invocation-validating-processing inside the view method.

For auth, I would port your scheme to Pyramid's built-in auth if feasable, because that will have better long-term support. I don't know whether AuthKit is compatible with Pyramid, and I believe AuthKit's author has stopped recommending it. If you have a complex permissions system, you'll have to decide whether the time it takes to port it to Pyramid's auth system is worth it. If you need authentication mechanisms that the built-in auth doesn't have, you might find them in repoze.who, but then you'll have to integrate the two (and we're still researching whether that is feasable). There's also repoze.what, which offers an authorization system with a permission hierarchy, but I don't see how it's any better than Pyramid's auth or AthKit.

If your application is using `app_globals` attributes, migrate them to Pyramid's `settings` dict. You can set them at the top of the main function, and access them in views as‘‘self.request.registry.settings[”my_setting”]‘‘, and in templates as `request.registry.settings["my_setting"]`.

Beaker caching is initialized in the settings. To use the cache decorators, see the following:

- http://docs.pylonsproject.org/projects/pyramid_beaker/dev/#beaker-cache-region-support
- http://beaker.groovie.org/caching.html#cache-regions

Pyramid has no cache object akin to Pylons `app_globals.cache`, but with the decorators you don't really need it. If you want to use it anyway, you can create a cache object by instantiating `beaker.cache.CacheManager`.

If you're using the REST/Atom URL structure (Routes `map.resource()` and "paster restcontroller"), there are no equivalent helpers in Pyramid at this time. You can define your own routes, or explore Pyramid's traversal feature. You can use route predicates to limit a route to a certain HTTP method. If you're tunneling PUT and DELETE via POST using the "_method" query parameter (as `webhelpers.html.tags` does), you can test the "_method" parameter directly with a route predicate: ‘request_param=”_method=PUT”’.

CHAPTER

TWELVE

API

APIs for functions and classes in the `akhet` package.

12.1 Config include function

`akhet.`**`includeme`**`(`*config*`)`

Add certain useful methods to a Pyramid `Configurator` instance.

Currently this adds the `.add_static_route()` method. (See `pyramid_sqla.static.add_static_route()`.)

12.2 Static route

`akhet.static.`**`add_static_route`**`(`*config*, *package*, *subdir*, *cache_max_age=3600*, ***add_route_args*`)`

Add a route and view to serve static files from a directory.

I create a catchall route that serves all URLs from a directory of static files if the corresponding file exists. Subdirectories are also handled. For example, the URL "/robots.txt" corresponds to file "PACKAGE/SUBDIR/robots.txt", and "/images/header/logo.png" corresponds to "PACKAGE/SUBDIR/images/header/logo.png". If the file doesn't exist, the route won't match the URL, and Pyramid will continue to the next route or traversal. The route name is 'static', which must not conflict with your application's other routes.

Usage in the application's __init__.py:

```
from akhet.static import add_static_route
add_static_route(config, "myapp", "static")
```

Or, more conveniently:

```
config.include("akhet")
config.add_static_route("myapp", "static")
```

This serves URLs from the "static" directory in package "myapp".

Arguments:

- `config`: a `pyramid.config.Configurator` instance.
- `package`: the name of the Python package containing the static files.

- `subdir`: the subdirectory in the package that contains the files. This should be a relative directory with '/' separators regardless of platform.
- `cache_max_age`: influences the `Expires` and `Max-Age` response headers returned by the view (default is 3600 seconds or five minutes).
- `**add_route_args`: additional arguments to `config.add_route`. 'name' defaults to "static" but can be overridden. (Every route in your application must have a unique name.) 'pattern' and 'view' may not be specified and will raise TypeError if they are.

12.3 URL generator

Convenience methods for generating URLs in templates.

I shadow various URL-generating functions in `pyramid.url` and attributes in the request. My main use is in templates where my method names are shorter than the originals and you don't have to pass a request argument. In Akhet, a subscriber callback registers an instance of me as `request.url_generator`, and another callback aliases me to the `url` global in templates. These callbacks are defined in *myapp/subscribers.py*.

The source code contains commented examples of "static_url" methods. We're not sure what's best here so we're leaving them commented and you can define them in a subclass however you wish. That's better than releasing a bad API which we'll then have to change and break existing apps.

Contributed by Michael Merickel. Modified by Mike Orr.

class `akhet.urlgenerator.`**`URLGenerator`**(*context*, *request*, *qualified=False*)

`__call__`(**elements*, ***kw*)
Same as the `.route` method.

`__init__`(*context*, *request*, *qualified=False*)
Instantiate a URLGenerator based on the current request.

- `request`: a Pyramid Request.
- `context`: a Pyramid Context.
- `qualified`: If true, return fully-qualified URLs with the "scheme://host" prefix. If false (default), return only the URL path if the underlying Pyramid function allows it.

`app`
The application URL or path.

I'm a "reified" attribute which means I start out as a property but I turn into an ordinary string attribute on the first access. This saves CPU cycles if I'm accessed often.

I return the application prefix of the URL. Append a slash to get the home page URL, or additional path segments to get a sub-URL.

If the constructor arg 'qualified' is true, I return `request.application_url`, otherwise I return `request.script_name`.

`ctx`
The URL of the default view for the current context.

I'm a "reified" attribute which means I start out as a property but I turn into an ordinary string attribute on the first access. This saves CPU cycles if I'm accessed often.

I am mainly used with traversal. I am different from `.app` when using context factories. I always return a qualified URL regardless of the constructor's 'qualified' argument.

route (*route_name*, **elements*, ***kw*)
Generate a route URL.

I return a URL based on a named route. Calling the URLGenerator instance is the same as calling me. If the constructor arg 'qualified' is true, I call `pyramid.url.route_url`, otherwise I call `pyramid.url.route_path`.

Arguments:

- `route_name`: the name of a route.
- `*elements`: additional segments to append to the URL path.

Keyword arguments are passed to the underlying function. The following are recognized:

- `_query`: the query parameters. May be a dict-like object with a `.items()` method or a sequence of 2-tuples.
- `_anchor`: the URL's "#ancor" fragment without the "#".
- `_app_url`: override the "scheme://host" prefix. (This also causes the result to be qualified if it wouldn't otherwise be.)
- Other keyword args override path variables defined in the route.

If the relevant route has a *pregenerator* defined, it may modify the elements or keyword args.

current (**elements*, ***kw*)
Generate a URL based on the current request's route.

I call `pyramid.url.current_route_url`. I'm the same as calling `.route` with the current route name. The result is always qualified regardless of the constructor's 'qualified' argument.

resource (**elements*, ***kw*)
Return a "resource URL" as used in traversal.

`*elements` is the same as with `.route`. Keyword args `query` and `anchor` are the same as the `_query` and `_anchor` args to `.route`.

When called without arguments, I return the same as `.ctx`.

12.4 MultiDict

MultiDict is defined in WebOb (`webobb.multidict.MultiDict`) but its documentation is so sparse that we have written our own documentation here. Pyramid's `request.params`, `request.GET` and `request.POST` are MultiDicts.

A MultiDict is like a regular dict except that each key can have multiple values. This is necessary to represent query parameters and form variables, which can be multiple. `request.params`, `request.GET` and `request.POST` are MultiDicts.

class `akhet.urlgenerator.`**MultiDict** (*dic=None*, ***kw*)
An ordered dict that can have multiple values for each key. All the regular dict methods are supported. "Ordered" means that `.keys()` will return the keys in the order they were defined.

Like a regular dict, the constructor can be called with an existing dict, an iterable of key-value tuples, or keyword args representing the dict keys. If the combination of args contain duplicate keys, all the impled values will be added to the dict, the keyword args last.

add (*key*, *value*)

Add the value to the dict, not overriding any previous values.

Note: `dic[key] = value` will replace all existing values for the key.

`getall`(*key*)
Return all values for the key as a list. The list will be empty if the key is not present in the dict.

Note: `dic[key]` and `dic.get(key)` return one arbitrary value, the last value added for the key.

`getone`(*key*)
Return exactly one value for the key. Raise `KeyError` if the key has zero values or more than one.

`mixed`()
Return a dict whose values are single values if the key appears once in the MultiDict, or lists if the key appears multiple times.

`dict_of_lists`()
Return a dict where each key is associated with a list of values.

`extend`(*dic=None*, ***kw*)
Add items to the MultiDict. The arguments are the same as the constructor's.

classmethod `view_list`(*lis*)
Create a dict that is a view on the given list

classmethod `from_fieldstorage`(*fs*)
Create a MultiDict from a `cgi.FieldStorage` instance

class `akhet.urlgenerator.`**`UnicodeMultiDict`**(*multi*, *encoding=None*, *errors="strict"*, *decode_keys=False*)
A MultiDict subclass that decodes returned values to unicode on the fly. Decoding is not applied to assigned values.

The key/value contents are assumed to be `str`/`strs` or `str`/`FieldStorages` (as is returned by the `paste.request.parse_` functions).

Can optionally also decode keys when the `decode_keys` argument is True.

`FieldStorage` instances are cloned, and the clone's `filename` variable is decoded. Its `name` variable is decoded when `decode_keys` is enabled.

class `akhet.urlgenerator.`**`NestedMultiDict`**(**dicts*)
A MultiDict subclass that wraps several MultiDict objects, treating them as one large MultiDict. A NestedMultiDict is read-only, although the contained MultiDicts are not.

class `akhet.urlgenerator.`**`TrackableMultiDict`**(*dic=None*, ***kw*)
This is a MultiDict that functions as an observable. It's used internally in WebOb where other parts of the API need to know when a GET or POST variable changes.

In addition to the regular MultiDict constructor args, you can pass `__tracker` (two leading underscores) which is a callback function, and `__name` which is the object's `repr()` name.

The callback function is called with zero to three positional args: `self`, `key`, and `value` (although the callback can name them differently, but it should give default values to all of them). `self` is the MultiDict's self. `key` is passed if the method deletes a particular key. `value` is passed if the method sets a particular key or adds an additional value to a key.

The `.copy` method returns a regular MultiDict. The tracker is not propagated to it.

There appears to be a bug in the source, that if you instantiate a TrackableMultiDuct without a tracker, it defaults to None and you'll get an exception when a routine tries to call it.

class `akhet.urlgenerator.`**`NoVars`**(*reason=None*)
This is an always-empty MultiDict. Adding an item item raises `KeyError`. It's used internally in WebOb to

distinguish between dicts that happen to be empty vs dicts that must be empty; e.g., `request.POST` in a GET request.

CHAPTER

THIRTEEN

OTHER PYRAMID FEATURES

Pyramid has several significant features that have no Pylons equivalent. These are one of the reasons the Pylons developers decided to switch to the repoze.BFG architecture (which is now Pyramid), so that we could leverage the existing code rather than having to write it from scratch.

13.1 Events

The events framework provides hooks where you can insert your own code into the request-processing sequence. It standardizes some features that were provided ad hoc in Pylons or not at all. To use it, you write a callback function that takes an `event` argument, and register it via `config.add_subscriber()` in the main function. Akhet applications have two predefined subscribers in *zzz/subscribers.py* which can serve as examples.

The events are listed on the Pyramid Events API page. Each has a different kind of `event` argument with different attributes.

- ApplicationCreated: called by `config.make_wsgi_app()` when the application starts up. `event.app` is the application instance.
- NewRequest: called at the beginning of each request, after the Request object is created. `event.request` is the request.
- ContextFound: called later than NewRequest, after the router has found the context object through URL dispatch or traversal. Use this if you need both the request and the context. `event.request` is the request, and `event.request.context` is the context.
- NewResponse: called after a view or its renderer returns a Response. `event.request` is the request, and `event.response` is the response.
- BeforeRender: called before rendering a template. `event` is a dict-like object containing the template's global variables. You can modify this dict to add new globals, but you'll get a `KeyError` if you try to set a key that already exists.

Each event type is a class in pyramid.events (e.g., `ApplicationCreated`). Each has a corresponding interface in pyramid.interfaces (e.g., `IApplicationCreated`). The class is what you pass as the second argument to `config.add_subscriber`. (You can also pass a dotted string name: "pyramid.events.ApplicationCreated".) The interface describes the API of the `event` object that's passed to your callback.

There are two other ways to modify or inspect responses, called "response callbacks" and "finished callbacks". These do not use the events infrastructure. They're documented in the Hooks page in the Pyramid manual. Unlike event subscribers, they have to be registered for each request. Note that response callbacks are NOT called if certain exceptions occur. Finished callbacks are always called, but they're called after the response has been sent to the user so they can't influence it.

The Pyramid manual says that NewResponse is not recommended and that middleware is better for modifying the response, but what it actually means is that it may be easier to write the equivalent functionality in middleware if you don't need Pyramid-specific data. On the other hand, if you want to log the response and certain request data in a database and you need Pyramid-specific data, an event or callback is suitable because you know right where the data is, whereas in middleware it may be difficult or impossible to get the data.

13.2 Extending applications

Pyramid provides ways to let you or another developer *extend* an application without touching its internal code. The second developer can create a separate Pyramid application that references the first one, and adds or overrides routes, views, templates, and static files. This allows the second developer to add functionality to the application or change the way it looks or behaves. The technique is described in the Extending chapter of the Pyramid manual.

However, Pyramid applications are not "pluggable" the way Django claims to be. That is, you can't expect two arbitrary Pyramid applications written by different people to fit together. Pyramid's flexibility makes this unfeasable. The developers would have to agree on a common set of conventions for structuring their applications, and write them with that in mind.

13.3 Request processing in detail

The Pyramid manual has a step-by-step list of how it processes a request, from the time it's received from the webserver to the time the response is given to the webserver. This is equivalent to the Pylons Execution Analysis, although it doesn't cover Paste's and the WSGI server's roles.

CHAPTER

FOURTEEN

FULL CHANGELOG

14.1 2.0b1 (unreleased)

- Revamp documentation to focus on Pyramid's 'alchemy' scaffold.
- Delete 'akhet' scaffold.

14.2 1.0.2 (2011-07-20)

- Adjust app skeleton to match URLGenerator.app fix in 1.0.1.

14.3 1.0.1 (2011-07-18)

- Fix bug in `URLGenerator.app`: it was returning the wrong value and was documented wrong.

14.4 1.0 (2011-04-04)

- App skeleton:
  - Simplify home page and add a Mako site template that can be easily extended by the user. New documentation chapters.
  - New default layout and stylesheet by Marcin Lulek (Ergo^), designed to be extensible and a learning tool.
  - Separate industry-standard "reset" stylesheet for cross-browser consistency.
  - Add "flash message" demo to home page.
  - Add "requirements" file for easy installation of dependencies.

14.5 1.0b2 (2011-03-19)

- App skeleton:
  - Add Beaker cache configuration
  - Fix bug in urlgenerator: missing variable assignment

14.6 1.0b1 (2011-03-19)

- Rename distribution to Akhet and app template to akhet.
- Delete all code pertaining to the SQLAlchemy library, which is now in the "SQLAHelper" package.
- `URLGenerator` makes generating route URLs and other application URLs more convenient.
- App template:
  - Change `handlers` to a package and refactor for larger applications.
  - Change `models` to a package.
  - Create a `lib` package and move helpers.py to it as Pylons does.
  - Add commented examples of advanced usages in init and base handler.
  - The `url` template global is now a URLGenerator instance. You can still call it as before to generate a route URL but don't pass the `request` arg any more. The URL generator is also available in views as `self.request.url_generator`.
  - Create the SQLAlchemy engine ourself; SQLAHelper no longer does this.
  - Change "[app:{{project}}]" to "[app:myapp]" in INI files so that the name is well known and easier to type on the command line (e.g., for 'pshell').
  - Ask whether to configure SQLAlchemy.
  - Switch to `pyramid_tm` transaction manager from `repoze.tm2`.
- 'akhet/tests/make_test_app.sh' is a quick-and-dirty script to create and run a test application.

14.7 Repository Akhet created

Repository "Akhet" was cloned from "pyramid_sqla" at this point. All tags "vVERSION" were renamed to "pyramid_sqla-VERSION". A new tag "pyramid_sqla-dev" points to the last code change before the split.

14.8 pyramid_sqla-dev (never released; changeset c0c74051c201)

- `add_static_route` is now a Pyramid config method if you call the new `includeme` function. This is used in the application template.
- Add `pyramid_sqla` as a dependency in the application template.
- Delete websetup.py. Console scripts are more flexible than "paster setup-app".
- Fix but that may have prevented create_db.py from finding the INI stanza.
- 100% test coverage contributed by Chris McDonough.
- Delete unneeded development code in static.py.
- Set Mako's 'strict_undefined' option in the application template.

14.9 pyramid_sqla-1.0rc1 (2010-01-26)

- 'pyramid_sqla' application template supports commit veto feature in repoze.tm2 1.0b1.
- Add production.ini to application template.
- Delete stray files in application template that were accidentally included.

14.10 pyramid_sqla-v0.2 (2011-01-19)

- Pyramid 1.0a10 spins off view handler support to 'pyramid_handlers' package.
- 'pyramid_sqla' application template depends on Pyramid>=1.0a10.

14.11 pyramid_sqla-0.1 (2011-01-12)

- Initial release.
- Warning: a change in Pyramid 1.0a10 broke applications created using the this version's application template. To run existing applications under Pyramid 1.0a10 and later, add a 'pyramid_handlers' dependency to the `requires` list in setup.py and reinstall the application.

CHAPTER

FIFTEEN

UNFINISHED: FRAGMENTS

15.1 INI files, logging, and pserve

15.1.1 development.ini

Some Pyramid add-ons also look here for configuration settings. For instance, Beaker looks for “cache.*” and “session.*” settings, and Mako looks for “mako.*” settings. You can use these to configure what kind of persistent session store to use, how long until idle sessions are discarded, and what character encoding to use for template output (the default is “utf-8”). possible options are listed in the Pyramid manual or the add-on’s manual.

You can of course create multiple INI files if you want to try the application out under different configuration scenarios, for instance to compare a database and a PostgresSQL database. You can even run them simultaneously as long as each configuration specifies a different port.

If you’re using Apache’s mod_proxy to proxy to a Python HTTP server, you might want to change this to “use = egg:pyramid#cherrypy”. This uses the CherryPy server, which is multithreaded like paste.httpserver (which Pylons and older versions of Pyramid used), but is more robust under high loads. (You’ll have to install CherryPy to use this.)

The Pyramid manual and Cookbook discuss other deployment scenarios like mod_wsgi and FastCGI.

I normally set “host = 127.0.0.1” and “port = 5000” after creating an application. That way it serves only request coming from the same computer rather than from any computer on the network. That enhances security when the debug toolbar is enabled. Port 5000 is the Pylons tradition, and it’s easier to remember and type than 6543.

“%(here)s” expands to the path of the directory containing the INI file.

15.1.2 production.ini

The default production file is just slightly different from the development file:

```
[app:main]
use = egg:Zzz

pyramid.reload_templates = false
pyramid.debug_authorization = false
pyramid.debug_notfound = false
pyramid.debug_routematch = false
pyramid.debug_templates = false
pyramid.default_locale_name = en
pyramid.includes = pyramid_tm

sqlalchemy.url = sqlite:///%(here)s/Zzz.db
```

```
[server:main]
use = egg:pyramid#wsgiref
host = 0.0.0.0
port = 6543
```

The most important difference here is that "pyramid_debugtoolbar" is NOT enabled. **This is vital for security!** Otherwise miscreants can type arbitrary Python commands in the interactive traceback if an exception occurs, and potentially read password files or damage the system.

If an exception occurs during a production request, the user will get a plain white error screen, "A server error occurred. Please contact the administrator." To customize that, see "Exception Views" in the Pyramid manual. The traceback will be dumped to the console, and will not be shown to the user. To customize how tracebacks are reported to the administrator, install the pyramid_exclog tween, which is covered below in Logging. (This replaces the WebError#error_catcher middleware which was used in Pylons and earlier versions of Pyramid.)

The debug settings are all set to false in production. This saves a few CPU cycles while it's processing requests.

The server section is unchanged from development.ini. The correct settings here depend on what webserver you're running the application with, so you'll have to configure this part yourself.

15.1.3 Logging

The bottom 2/3 of both INI files contain several sections to configure Python's logging system. This is the same as in Pylons. We can't explain the entire logging syntax here, but these are the sections most often customized by users:

```
[logger_root]
level = INFO
handlers = console

[logger_zzz]
level = DEBUG
handlers =
qualname = zzz

[logger_sqlalchemy]
level = INFO
handlers =
qualname = sqlalchemy.engine
# "level = INFO" logs SQL queries.
# "level = DEBUG" logs SQL queries and results.
# "level = WARN" logs neither.  (Recommended for production systems.)
```

These define a logger "root", "zzz" (the application's package name), and "sqlalchemy.engine" (specified in the qualname). Each has a 'level' variable which can be DEBUG, INFO, WARN, ERROR, or CRITICAL. Each level also logs the levels on its right, so WARN logs warnings and errors. Logger names are in a dotted hierarchy, so that "sqlalchemy.engine" affects all loggers below it ("sqlalchemy.engine.ENGINE1", etc). "root" affects all loggers that aren't otherwise specified.

Generally, DEBUG is debugging information, INFO is chatty success messages, WARN means something might be wrong, ERROR means something is definitely wrong, and CRITICAL means you'd better fix it now or else. But there's nothing to enforce this, so each library chooses how to log things. So SQLAlchemy logs SQL queries at the INFO level on "sqlalchemy.engine.ENGINE_NAME", even though some people would consider this debugging information.

Logger names do NOT automatically correspond to Python module names, although it's customary to do so if there's no better name for the logger. You can do this by putting the following at the top of each module:

> import logging log = logging.getLogger(__name__)

This creates a variable `log` which is a logger named after the module. So if the module is `zzz.views`, the logger is "zzz.views".

The default *development.ini* displays all messages from the application modules (logger_zzz = DEBUG). It displays SQL queries (logger_sqlalchemy = INFO). It displays other messages only if they're warnings or above (logger_root = WARN). The default *production.ini* sets all these to WARN, so it will not log anything except warnings or errors.

You can create additional loggers by adding a "[logger_yourname]" section, listing it in the "[loggers]" section, and calling `logging.getLogger("yourname")` in the Python module.

To activate logging in a utility script the way "pserve" does, do the following:

```
import logging.config
logging.config.fileConfig(INI_FILENAME)
```

15.2 Init module and main function

a dict based on the "[GLOBAL]" section of the INI file.

```
# AKHET 1

 from pyramid.config import Configurator
 import akhet
 import pyramid_beaker
 import sqlahelper
 import sqlalchemy

 def main(global_config, XXsettings):
     """ This function returns a Pyramid WSGI application.
     """

     # Here you can insert any code to modify the ``settings`` dict.
     # You can:
     # * Add additional keys to serve as constants or "global variables" in the
     #   application.
     # * Set default values for settings that may have been omitted.
     # * Override settings that you don't want the user to change.
     # * Raise an exception if a setting is missing or invalid.
     # * Convert values from strings to their intended type.

     # Create the Pyramid Configurator.
     config = Configurator(settings=settings)
     config.include("pyramid_handlers")
     config.include("akhet")

     # Initialize database
     engine = sqlalchemy.engine_from_config(settings, prefix="sqlalchemy.")
     sqlahelper.add_engine(engine)
     config.include("pyramid_tm")

     # Configure Beaker sessions
     session_factory = pyramid_beaker.session_factory_from_settings(settings)
     config.set_session_factory(session_factory)

     # Configure renderers and event subscribers
     config.add_renderer(".html", "pyramid.mako_templating.renderer_factory")
     config.add_subscriber("zzz.subscribers.create_url_generator",
         "pyramid.events.ContextFound")
     config.add_subscriber("zzz.subscribers.add_renderer_globals",
                           "pyramid.events.BeforeRender")

     # Set up view handlers
     config.include("zzz.handlers")

     # Set up other routes and views
     # ** If you have non-handler views, create create a ``zzz.views``
     # ** module for them and uncomment the next line.
     #
     #config.scan("zzz.views")

     # Mount a static view overlay onto "/". This will serve, e.g.:
     # ** "/robots.txt" from "zzz/static/robots.txt" and
     # ** "/images/logo.png" from "zzz/static/images/logo.png".
     #
     config.add_static_route("zzz", "static", cache_max_age=3600)

     # Mount a static subdirectory onto a URL path segment.
     # ** This not necessary when using add_static_route above, but it's the
     # ** standard Pyramid way to serve static files under a URL prefix (but
     # ** not top-level URLs such as "/robots.txt"). It can also serve files from
     # ** third-party packages, or point to an external HTTP server (a static
     # ** media server).
     # ** The first commented example serves URLs under "/static" from the
     # ** "zzz/static" directory. The second serves URLs under
     # ** "/deform" from the third-party ``deform`` distribution.
     #
     #config.add_static_view("static", "zzz:static")
     #config.add_static_view("deform", "deform:static")

     return config.make_wsgi_app()
```

(Note: `**settings` in line 7 is displayed as `XXsettings` due to a limitation in our documentation generator: "`*`" in code blocks outside comments make Vim's syntax highlighting go bezerk.)

Lines 11-18 are a long comment explaining how you can modify the `settings` dict. If you have any code to set "global variables" for the application, or to validate the settings or convert the values from strings to other types, put the code here. (We're considering a default routine to validate the settings but haven't decided whether to use homegrown code, Colander, FormEncode, or another validation library.)

Line 21 instantiates a `Configurator` which will create the application. (It's not the application itself.) Lines 22-23 add plug-in functionality to the configurator. The argument is the name of a module that contains an `includeme()` function. Line 22 ultimately creates the `config.add_handler()` method; line 23 creates the `config.add_static_route()` method.

Line 26 creates a SQLAlchemy engine based on the "sqlalchemy.url" setting in *development.ini*. The default setting is "sqlite:///%(here)s/db.sqlite", which creates or opens a database "db.sqlite" in the same directory as the INI file. You can also pass other engine arguments to SQLAlchemy, either by putting them in the INI file with the "sqlalchemy." prefix, or by passing them as keyword args. Line 27 adds the engine to the `sqlahelper` library so that the model can use it; it also updates the library's contextual session. Line 28 initializes the "pyramid_tm" transaction manager. SQLAHelper is further explained in the Models section below; the transaction manager is explained in the "Transaction Manager" chapter.

(Note: if you answered 'n' to the SQLAlchemy question when creating the application, lines 4-5 and 25-28 will not be present in your module.)

Lines 31-32 configure the session factory.

Line 35 tells Pyramid to render **.html* templates using Mako. Pyramid out of the box renders Mako templates with the **.mako* or **.mak* extensions, and Chameleon templates with the **.pt* extension, but you have to tell it if you want to use a different extension or another template engine. Third-party packages are available for using Jinja2 (`pyramid_jinja2`), and a Genshi emulator using Chameleon (`pyramid_genshi_chameleon`),

Lines 36-39 registers event subscribers, which are callback functions called at specific points during request processing. Lines 36-37 register a callback that instantiates a URL generator (described in the Templates section below and in the API chapter). Lines 38-39 register a callback which adds several Pylons-like variables to the template namespace whenever a template is rendered. The callbacks are defined in the `zzz.subscribers` module, which you can modify.

Lines 42 configures routing. Actually it calls an include function in the handlers package. We'll explore routing more fully later.

Lines 44-48 and 56-67 are commented code; they show how to enable certain advanced features.

Line 54 is equivalent to the *public* directory in Pylons applications. It's not a standard part of Pyramid, which handles static files a different way, but this method is closer to the Pylons tradition. Any URLs which did not match a dynamic route will be compared to the contents of the *zzz/static* directory, and if a file exists for the URL, it is served. Unlike Pylons, this happens after the dynamic routes are tried rather than before. This means that any dynamic route that might accidentally match a static resource must explicitly exclude that URL.

This is just one of several ways to serve static files in Pyramid, each with its own advantages and disadvantages. These are all discussed below in the Static Files section.

Line 69 creates and returns a Pyramid WSGI application based on the configuration.

This short main function – compared to Pylons' three functions in three modules – allows an entire small application to be defined in a single module. Half the lines are comments so they can be deleted. A short main function is useful for small demos, but the principle also leads to a different developer culture. Pylons' application skeleton is complex enough that most people don't stray from it, and Pylons' documentation emphasizes using "paster serve" rather than other invocation methods. Pyramid's docs encourage users to structure everything outside `main()` as they wish, and they describe "paster serve" as just one way to invoke the application. The INI files and "paster serve" are just for your convenience; you don't have to use them.

15.2.1 A bit more about Paster

"paster serve" does several other things besides calling the main function. It interpolates "%(here)s" placeholders in the INI file, as well as variables in the "[DEFAULT]" section (which we aren't using here). It configures logging, and finds the application by looking up the entry point specified in the 'use' variable. All this can be done by the following code in both Pyramid and Pylons:

```
import logging.config
import os
import paste.deploy.loadwsgi as loadwsgi
ini_path = "/path/to/development.ini"
logging.config.fileConfig(ini_path)
app_dir, ini_file = os.path.split(ini_path)
app = loadwsgi.loadapp("config:" + ini_file, relative_to=app_dir)
```

15.3 Models

The default *zzz/models/__init__.py* looks like this:

```
import logging
import sqlahelper
import sqlalchemy as sa
import sqlalchemy.orm as orm
import transaction

log = logging.getLogger(__name__)

Base = sqlahelper.get_base()
Session = sqlahelper.get_session()


#class MyModel(Base):
#    __tablename__ = "models"
#
#    id = sa.Column(sa.Integer, primary_key=True)
#    name = sa.Column(sa.Unicode(255), nullable=False)
```

Pylons applications have a "zzz.model.meta" model to hold SQLAlchemy's housekeeping objects, but Akhet uses the SQLAHelper library which holds them instead. This gives you more freedom to structure your models as you wish, while still avoiding circular imports (which would happen if you defined Session in the main module and then import the other modules into it; the other modules would import the main module to get the Session, and voilà circular imports).

A real application would replace the commented `MyModel` class with one or more ORM classes. The example uses SQLAlchemy's "declarative" syntax, although of course you don't have to.

15.3.1 SQLAHelper

The SQLAHelper library is a holding place for the application's contextual session (normally assigned to a `Session` variable with a capital S, to distinguish it from a regular SQLAlchemy session), all engines used by the application, and an optional declarative base. We initialized it via the `sqlahelper.add_engine` line in the main function. Because we did not specify an engine name, the library set the engine name to "default", and also bound the contextual session and the base's metadata to it.

There's not much else to know about SQLAHelper. You can call `get_session()` at any time to get the contextual session. You can call `get_engine()` or `get_engine(name)` to retrieve an engine that was previously added. You can call `get_base()` to get the declarative base.

If you need to modify the session-creation parameters, you can call `get_session().config(...)`. But if you modify the session extensions, see the "Transaction Manager" chapter to avoid losing the extension that powers the transaction manager.

15.4 View handlers

The default *zzz.handlers* package contains a *main* module which looks like this:

```
import logging

from pyramid_handlers import action

import zzz.handlers.base as base
import zzz.models as model

log = logging.getLogger(__name__)
```

```
class Main(base.Handler):
    @action(renderer="index.html")
    def index(self):
        log.debug("testing logging; entered Main.index()")
        return {"project":"Zzz"}
```

This is clearly different from Pylons, and the `@action` decorator looks a bit like TurboGears. The decorator has three optional arguments:

name

The action name, which is the target of the route. Normally this is the same as the view method name but you can override it, and you must override it when stacking multiple actions on the same view method.

renderer

A renderer name or template filename (whose extension indicates the renderer). A renderer converts the view's return value into a Response object. Template renderers expect the view to return a dict; other renderers may allow other types. Two non-template renderers are built into Pyramid: "json" serializes the return value to JSON, and "string" calls `str()` on the return value unless it's already a Unicode object. If you don't specify a renderer, the view must return a Response object (or any object having three particular attributes described in Pyramid's Response documentation). In all cases the view can return a Response object to bypass the renderer. HTTP errors such as HTTPNotFound also bypass the renderer.

permission

A string permission name. This is discussed in the Authorization section below.

The Pyramid developers decided to go with the return-a-dict approach because it helps in two use cases:

1. Unit testing, where you want to test the data calculated rather than parsing the HTML output. This works by default because `@action` itself does not modify the return value or arguments; it merely sets function attributes or interacts with the configurator.

2. Situations where several URLs render the same data using different templates or different renderers (like "json"). In that case, you can put multiple `@action` decorators on the same method, each with a different name and renderer argument.

Two functions in `pyramid.renderers` are occasionally useful in views:

`pyramid.renderers.`**`render`**`(`*renderer_name*, *value*, *request=None*, *package=None*`)`
Render a template and return a string. 'renderer_name' is a template filename or renderer name. 'value' is a dict of template variables. 'request' is the request, which is needed only if the template cares about it.

If the function can't find the template, try passing "zzz:templates/" as the `package` arg.

`pyramid.renderers.`**`render_to_response`**`(`*renderer_name*, *value*, *request=None*, *package=None*`)`
Render a template, instantiate a Response, set the Response's body to the result of the rendering, and return the Response. The arguments are the same as for `render()`, except that 'request' is more important.

The handler class inherits from a base class defined in *zzz.handlers.base*:

```
"""Base classes for view handlers.
"""

class Handler(object):
    def __init__(self, request):
        self.request = request

        #c = self.request.tmpl_context
        #c.something_for_site_template = "Some value."
```

Pyramid does not require a base class but Akhet defines one for convenience. All handlers should set `self.request` in their `.__init__` method, and the base handler does this. It also provides a place to put common methods used by several handler classes, or to set `tmpl_context` (`c`) variables which are used by your site template (common to all views or several views). (You can use `c` in view methods the same way as in Pylons, although this is not recommended.)

Note that non-template renders such as "json" ignore `c` variables, so it's really only useful for HTML-only data like which stylesheet to use.

The routes are defined in *zzz/handlers/__init__.py*:

```
"""View handlers package.
"""

def includeme(config):
    """Add the application's view handlers.
    """
    config.add_handler("home", "/", "zzz.handlers.main:Main",
                       action="index")
    config.add_handler("main", "/{action}", "zzz.handlers.main:Main",
        path_info=r"/(?!favicon\.ico|robots\.txt|w3c)")
```

`includeme` is a configurator "include" function, which we've already seen. This function calls `config.add_handler` twice to create two routes. The first route connects URL "/" to the `index` view in the `Main` handler.

The second route connects all other one-segment URLs (such as "/hello" or "/help") to a same-name method in the `Main` handler. "{action}" is a path variable which will be set the corresponding substring in the URL. Pyramid will look for a method in the handler with the same action name, which can either be the method's own name or another name specified in the 'name' argument to `@action`. Of course, these other methods ("hello" and "help") don't exist in the example, so Pyramid will return 400 Not Found status.

The 'path_info' argument is a regex which excludes certain URLs from matching ("/favicon.ico", "/robots.txt", "/w3c"). These are static files or directories that would syntactically match "/{action}", but we want these to go to a later route instead (the static route). So we set a 'path_info' regex that doesn't match them.

15.4.1 Redirecting and HTTP errors

To issue a redirect inside a view, return an HTTPFound:

```
from pyramid.httpexceptions import HTTPFound

def myview(self):
    return HTTPFound(location=request.route_url("foo"))
    # Or to redirect to an external site
    return HTTPFound(location="http://example.com/")
```

You can return other HTTP errors the same way: `HTTPNotFound`, `HTTPGone`, `HTTPForbidden`, `HTTPUnauthorized`, `HTTPInternalServerError`, etc. These are all subclasses of both `Response` and `Exception`. Although you can raise them, Pyramid prefers that you return them instead. If you intend to raise them, you have to define an exception view that receives the exception argument and returns it, as shown in the Views chapter in the Pyramid manual. (On Python 2.4, you also have to call the instance's `.exception()` method and raise that, because you can't raise instances of new-style classes in 2.4.) A future version of Pyramid may have an exception view built-in; this would conflict with your exception view so you'd need to delete it, but there's no need to worry about that until/if it actually happens.

Pyramid catches two non-HTTP exceptions by default, `pyramid.exceptions.NotFound` and `pyramid.exceptions.Forbidden`, which it sends to the Not Found View and the Forbidden View respectively. You can override these views to display custom HTML pages.

15.5 More on routing and traversal

15.5.1 Routing methods and view decorators

Pyramid has several routing methods and view decorators. The ones we've seen, from the `pyramid_handlers` package, are:

`@action(**kw)`
: I make a method in a class into a *view* method, which `config.add_handler` can connect to a URL pattern. By definition, any class that contains view methods is a view handler. My most interesting args are 'name' and 'renderer'. If 'name' is NOT specified, the action name is the same as the method name. If 'name' IS specified, the action name can be different. If 'renderer' is specified, it indicates a renderer or template (and the template's extension indicates a renderer). If multiple `@action` decorators are put on a single method, each must have a different name, and they presumably will have different renderers too.

`config.`**`add_handler`**`(`*name*, *pattern*, *handler*, *action=None*, ***kw*`)`
: I create a route connecting the URL pattern to the handler class. If 'action' is specified, I connect the route to that specific action (a method decorated with the `@action` decorator). If 'action' is not specified, the pattern must contain a "{action}" placeholder. In that case I scan the handler class for all possible actions. It is an error to specify both "{action}" and an `action` arg. I pass extra keyword args to `config.add_route`, and keyword args in the `@action` decorator to `config.add_view`.

`config.add_handler` calls two lower-level methods which you can also call directly:

`config.`**`add_route`**`(`*name*, *pattern*, ***kw*`)`
: Create a route connecting a URL pattern directly to a view callable outside a handler. The view is specified with a 'view' arg. If the view is a function, it must take a Request argument and return a Response (or any object with the three required attributes). If it's a class, the constructor takes the Request argument and the specified method (`.__call__` by default) is called with no arguments.

`config.`**`add_view`**`(`***kw*`)`
: I register a view (specified with a 'view' arg). In URL dispatch, you normally don't call this directly but let `config.add_handler` or `config.add_route` call it for you. In traversal, you call this to register a view. The 'name' argument is the view name, which is used by traversal to choose which view to invoke.

Two others you should know about:

`config.`**`scan`**`(`*package=None*`)`
: I scan the specified package (which may be an asset spec) and import all its modules recursively, looking for functions decorated with `@view_config`. For each such function, I call `add_view` passing the decorator's args to it. I can also scan a package, in which case all submodules in the package are recursively scanned. If no package is specified, I scan the caller's package (i.e., the entire application).

  I can also be called for my side effect of importing all of a package's modules even if none of them contain `@view_config`.

`@view_config(**kw)`
: I decorate a function so that `config.scan` will recognize it as a view callable, and I also hold `add_view` arguments that `config.scan` will pick up and apply. I can also decorate a class or a method in a class.

15.5.2 Route arguments and predicates

`config.add_handler` accepts a large number of keyword arguments. We'll list the ones most commonly used with Pylons-like applications here. For full documentation see the add_route API. Most of these arguments can also be used with `config.add_route`.

The arguments are divided into *predicate arguments* and *non-predicate arguments*. Predicate arguments determine whether the route matches the current request: all predicates must pass in order for the route to be chosen. Non-predicate arguments do not affect whether the route matches.

name

> [Non-predicate] The first positional arg; required. This must be a unique name for the route, and is used in views and templates to generate the URL.

pattern

> [Predicate] The second positional arg; required. This is the URL path with optional "{variable}" placeholders; e.g., "/articles/{id}" or "/abc/{filename}.html". The leading slash is optional. By default the placeholder matches all characters up to a slash, but you can specify a regex to make it match less (e.g., "{variable:d+}" for a numeric variable) or more ("{variable:.*}" to match the entire rest of the URL including slashes). The substrings matched by the placeholders will be available as *request.matchdict* in the view.
>
> A wildcard syntax "*varname" matches the rest of the URL and puts it into the matchdict as a tuple of segments instead of a single string. So a pattern "/foo/{action}/*fizzle" would match a URL "/foo/edit/a/1" and produce a matchdict `{'action': u'edit', 'fizzle': (u'a', u'1')}`.
>
> Two special wildcards exist, "*traverse" and "*subpath". These are used in advanced cases to do traversal on the right side of the URL, and should be avoided otherwise.

factory

> [Non-predicate] A callable (or asset spec). In URL dispatch, this returns a *root resource* which is also used as the *context*. If you don't specify this, a default root will be used. In traversal, the root contains one or more resources, and one of them will be chosen as the context.

xhr

> [Predicate] True if the request must have an "X-Reqested-With" header. Some Javascript libraries (JQuery, Prototype, etc) set this header in AJAX requests.

request_method

> [Predicate] An HTTP method: "GET", "POST", "HEAD", "DELETE", "PUT". Only requests of this type will match the route.

path_info

> [Predicate] A regex compared to the URL path (the part of the URL after the application prefix but before the query string). The URL must match this regex in order for the route to match the request.

request_param

> [Predicate] If the value doesn't contain "=" (e.g., "q"), the request must have the specified parameter (a GET or POST variable). If it does contain "=" (e.g., "name=value"), the parameter must have the specified value.
>
> This is especially useful when tunnelling other HTTP methods via POST. Web browsers can't submit a PUT or DELETE method via a form, so it's customary to use POST and to set a parameter `_method="PUT"`. The framework or application sees the "_method" parameter and pretends the other HTTP method was requested. In Pyramid you can do this with `request_param="_method=PUT`.

header

[Predicate] If the value doesn't contain ":"; it specifies an HTTP header which must be present in the request (e.g., "If-Modified-Since"). If it does contain ":", the right side is a regex which the header value must match; e.g., "User-Agent:Mozilla/.*". The header name is case insensitive.

accept

[Predicate] A MIME type such as "text/plain", or a wildcard MIME type with a star on the right side ("text/*") or two stars ("*/*"). The request must have an "Accept:" header containing a matching MIME type.

custom_predicates

[Predicate] A sequence of callables which will be called in order to determine whether the route matches the request. The callables should return `True` or `False`. If any callable returns `False`, the route will not match the request. The callables are called with two arguments, `info` and `request`. `request` is the current request. `info` is a dict which contains the following:

```
info["match"]  =>  the match dict for the current route
info["route"].name  =>  the name of the current route
info["route"].pattern  =>  the URL pattern of the current route
```

Use custom predicates argument when none of the other predicate args fit your situation. See <http://docs.pylonsproject.org/projects/pyramid/1.0/narr/urldispatch.html#custom-route-predicates>‘ in the Pyramid manual for examples.

You can modify the match dict to affect how the view will see it. For instance, you can look up a model object based on its ID and put the object in the match dict under another key. If the record is not found in the model, you can return False to prevent the route from matching the request; this will ultimately case HTTPNotFound if no other route or traversal matches the URL. The difference between doing this and returning HTTPNotFound in the view is that in the latter case the following routes and traversal will never be consulted. That may or may not be an advantage depending on your application.

15.5.3 View arguments

The 'name', 'renderer' and 'permission' arguments described for `@action` can also be used with `@view_config` and `config.add_view`.

`config.add_route` has counterparts to some of these such as 'view_permission'.

`config.add_view` also accepts a 'view' arg which is a view callable or asset spec. This arg is not useful for the decorators which already know the view.

The 'wrapper' arg can specify another view, which will be called when this view returns. (XXX Is this compatible with view handlers?)

15.6 The request object

The Request object contains all information about the current request state and application state. It's available as `self.request` in handler views, the `request` arg in view functions, and the `request` variable in templates. In pshell or unit tests you can get it via `pyramid.threadlocal.get_current_request()`. (You shouldn't use the threadlocal back door in most other cases. If something you call from the view requires it, pass it as an argument.)

Pyramid's `Request` object is a subclass of WebOb Request just like 'pylons.request' is, so it contains all the same attributes in methods like `params`, `GET`, `POST`, `headers`, `method`, `charset`, `date`, `environ`, `body`, and body_file. The most commonly-used attribute is request.params, which is the query parameters and POST variables.

Pyramid adds several attributes and methods. context, matchdict, matched_route, registry, registry.settings, session, and tmpl_context access the request's state data and global application data. route_path, route_url, resource_url, and static_url generate URLs, shadowing the functions in pyramid.url. (One function, current_route_url, is available only as a function.)

Rather than repeating the existing documentation for these attributes and methods, we'll just refer you to the original docs:

- Pyramd Request, Response, HTTP Exceptions, and MultiDict
- Pyramid Request API
- WebOb Request API
- Pyramid Response API
- WebOb Response API

MultiDict is not well documented so we've written our own MultiDict API page. In short, it's a dict-like object that can have multiple values for each key. request.params, request.GET, and request.POST are MultiDicts.

Pyramid has no pre-existing Response object when your view starts executing. To change the response status type or headers, you can either instantiate your own pyramid.response.Response object and return it, or use these special Request attributes defined by Pyramid:

```
request.response_status = "200 OK"
request.response_content_type = "text/html"
request.response_charset = "utf-8"
request.response_headerlist = [
    ('Set-Cookie', 'abc=123'), ('X-My-Header', 'foo')]
request.response_cache_for = 3600    # Seconds
```

Akhet adds one Request attribute. request.url_generator, which is used to implement the url template global described below.

15.7 Templates

Pyramid has built-in support for Mako and Chameleon templates. Chameleon runs only on CPython and Google App Engine, not on Jython or other platforms. Jinja2 support is available via the pyramid_jinja2 package on PyPI, and a Genshi emulator using Chameleon is in the pyramid_chameleon_genshi package.

Whenever a renderer invokes a template, the template namespace includes all the variables in the view's return dict, plus the following:

request
The current request.

context
The context (same as request.context).

renderer_name
The fully-qualified renderer name; e.g., "zzz:templates/foo.mako".

renderer_info
An object with attributes name, package, and type.

The subscriber in your application adds the following additional variables:

c, tmpl_context
: `request.tmpl_context`

h
: The helpers module, defined as "zzz.helpers". This is set by a subscriber callback in your application; it is not built into Pyramid.

session
: `request.session`.

url
: In Akhet, a URLGenerator object. In Pyramid's built-in application templates that use URL dispatch, an alias to the `route_url` *function*, which requires you to pass the route name as the first arg and the request as the second arg.

  The URLGenerator object has convenience methods for generating URLs based on your application's routes. See the complete list on the API page.

  By default the generator creates unqualified URLs (i.e., without the "scheme://hostname" prefix) if the underlying Pyramid functions allow it. To get absolute URLs throughout the application, edit *zzz/subscribers.py*, go to the line where the URLGenerator is instantiated, and change the 'qualified' argument to True. Pylons traditionally uses unqualified URLs, while Pyramid traditionally uses qualified URLs. Note that qualified URLs may be wrong if the application is running behind a reverse proxy! (E.g., Apache's mod_proxy.) The generated URL may be "http://localhost:5000" which is correct for the application but invalid to the end user (who needs the proxy's URL, "https://example.com").

15.7.1 Advanced template usage

If you need to fill a template within view code or elsewhere, do this:

```
from pyramid.renderers import render
variables = {"foo": "bar"}
html = render("mytemplate.mako", variables, request=request)
```

There's also a `render_to_response` function which invokes the template and returns a Response, but usually it's easier to let `@action` or `@view_config` do this. However, if your view has an if-stanza that needs to override the template specified in the decorator, `render_to_response` is the way to do it.

```
@action(renderer="index.html")
def index(self):
    records = models.MyModel.all()
    if not records:
        return render_to_response("no_records.html")
    return {"records": records}
```

For further information on templating see the Templates section in the Pyramid manual, the Mako manual, and the Chameleon manual. You can customize Mako's TemplateLookup by setting "mako.*" variables in the INI file.

15.7.2 Site template

Most applications using Mako will define a site template something like this:

```
<!DOCTYPE html>
<html>
  <head>
    <title>${self.title()}</title>
    <link rel="stylesheet" href="${application_url}/default.css"
```

```
        type="text/css" />
  </head>
  <body>

<!-- *** BEGIN page content *** -->
${self.body()}
<!-- *** END page content *** -->
  </body>
</html>
<%def name="title()" />
```

Then the page templates can inherit it like so:

```
<%inherit file="/site.html" />
<%def name="title()">My Title</def>
... rest of page content goes here ...
```

15.8 Static files

Pyramid has five ways to serve static files. Each way has different advantages and limitations, and requires a different way to generate static URLs.

`config.add_static_route`

This is the Akhet default, and is closest to Pylons. It serves the static directory as an overlay on "/", so that URL "/robots.txt" serves "zzz/static/robots.txt", and URL "/images/logo.png" serves "zzz/static/images/logo.png". If the file does not exist, the route will not match the URL and Pyramid will try the next route or traversal. You cannot use any of the URL generation methods with this; instead you can put a literal URL like "${application_url}/images/logo.png" in your template.

Usage:

```
config.include('akhet')
config.add_static_route('zzz', 'static', cache_max_age=3600)
# Arg 1 is the Python package containing the static files.
# Arg 2 is the subdirectory in the package containing the files.
```

`config.add_static_view`

This is Pyramid's default algorithm. It mounts a static directory under a URL prefix such as "/static". It is not an overlay; it takes over the URL prefix completely. So URL "/static/images/logo.png" serves file "zzz/static/images/logo.png". You cannot serve top-level static files like "/robots.txt" and "/favicon.ico" using this method; you'll have to serve them another way.

Usage:

```
config.add_static_view("static", "zzz:static")
# Arg 1 is the view name which is also the URL prefix.
# It can also be the URL of an external static webserver.
# Arg 2 is an asset spec referring to the static directory/
```

To generate "/static/images/logo.png" in a Mako template, which will serve "zzz/static/images/logo.png":

```
href="${request.static_url('zzz:static/images/logo.png')}
```

One advantage of add_static_view is that you can copy the static directory to an external static webserver in production, and static_url will automatically generate the external URL:

```
# In INI file
static_assets = "static"
# -OR-
static_assets = "http://staticserver.com/"

config.add_static_view(settings["static_assets"], "zzz:static")

href="${request.static_url('zzz:static/images/logo.png')}"
## Generates URL "http://staticserver.com/static/images/logo.png"
```

Other ways

> There are three other ways to serve static files. One is to write a custom view callable to serve the file; an example is in the Static Assets section of the Pyramid manual. Another is to use `paste.fileapp.FileApp` or `paste.fileapp.DirectoryApp` in a view. (More recent versions are in the "PasteOb" distribution.) These three ways can be used with `request.route_url()` because the route is an ordinary route. The advantage of these three ways is that they can serve a static file or directory from a normal view callable, and the view can be protected separately using the usual authorization mechanism.

15.9 Session, flash messages, and secure forms

Pyramid's session object is `request.session`. It has its own interface but uses Beaker on the back end, and is configured in the INI file the same way as Pylons' session. It's a dict-like object and can store any pickleable value. It's pulled from persistent storage only if it's accessed during the request processing, and it's (re)saved only if the data changes.

Unlike Pylons' sesions, you don't have to call `session.save()` after adding or replacing keys because Pyramid does that for you. But you do have to call `session.changed()` if you modify a mutable value in place (e.g., a session value that's a list or dict) because Pyramid can't tell that child objects have been modified.

You can call `session.invalidate()` to discard the session data at the end of the request. `session.created` is an integer timestamp in Unix ticks telling when the session was created, and `session.new` is true if it was created during this request (as opposed to being loaded from persistent storage).

Pyramid sessions have two extra features: flash messages and a secure form token. These replace `webhelpers.pylonslib.flash` and `webhelpers.pylonslib.secure_form`, which are incompatible with Pyramid.

Flash messages are a session-based queue. You can push a message to be displayed on the next request, such as before redirecting. This is often used after form submissions, to push a success or failure message before redirecting to the record's main screen. (This is different from form validation, which normally redisplays the form with error messages if the data is rejected.)

To push a message, call `request.session.flash("My message.")` The message is normally text but it can be any object. Your site template will then have to call `request.session.pop_flash()` to retrieve the list of messages, and display then as it wishes, perhaps in <div>'s or a <ul>. The queue is automatically cleared when the messages are popped, to ensure they are displayed only once.

The full signature for the flash method is:

```
session.flash(message, queue='', allow_duplicate=True)
```

You can have as many message queues as you wish, each with a different string name. You can use this to display warnings differently from errors, or to show different kinds of messages at different places on the page. If `allow_duplicate` is false, the message will not be inserted if an identical message already exists in that queue. The `session.pop_flash` method also takes a queue argument to specify a queue. You can also use `session.peek_flash` to look at the messages without deleting them from the queue.

The secure form token prevents cross-site request forgery (CSRF) attacts. Call `session.get_csrf_token()` to get the session's token, which is a random string. (The first time it's called, it will create a new random token and store it in the session. Thereafter it will return the same token.) Put the token in a hidden form field. When the form submission comes back in the next request, call `session.get_csrf_token()` again and compare it to the hidden field's value; they should be the same. If the form data is missing the field or the value is different, reject the request, perhaps by returning a forbidden status. `session.new_csrf_token()` always returns a new token, overwriting the previous one if it exists.

15.10 WebHelpers and forms

Most of WebHelpers works with Pyramid, including the popular `webhelpers.html` subpackage, `webhelpers.text`, and `webhelpers.number`. Pyramid does not depend on WebHelpers so you'll have to add the dependency to your application if you want to use it. The only part that doesn't work with Pyramid is the `webhelpers.pylonslib` subpackage, which depends on Pylons' special globals.

We are working on a form demo that compares various form libraries: Deform, Formish, FormEncode/htmlfill.

To organize the form display-validate-action route, we recommend the `pyramid_simpleform` package. It replaces `@validate` in Pylons. It's not a decorator because too many people found the decorator too inflexible: they ended up copying part of the code into their action method.

WebHelpers 1.3 has some new URL generator classes to make it easier to use with Pyramid. See the `webhelpers.paginate` documentation for details. (Note: this is *not* the same as Akhet's URL generator; it's a different kind of class specifically for the paginator's needs.)

15.11 Shell

paster pshell is similar to Pylons' "paster shell". It gives you an interactive shell in the application's namespace with a dummy request. Unlike Pylons, you have to specify the application section on the command line because it's not "main". Akhet, for convenience, names the section "myapp" regardless of the actual application name.

```
$ paster pshell development.ini myapp
Python 2.6.6 (r266:84292, Sep 15 2010, 15:52:39)
[GCC 4.4.5] on linux2
Type "help" for more information. "root" is the Pyramid app root object, "registry" is the Pyramid re
>>> registry.settings["sqlalchemy.url"]
'sqlite:////home/sluggo/exp/pyramid-docs/main/workspace/Zzz/db.sqlite'
>>> import pyramid.threadlocal
>>> request = pyramid.threadlocal.get_current_request()
>>>
```

As the example above shows, the interactice namespace contains two objects initially: `root` which is the root object, and `registry` from which you can access the settings. To get the request, you have to use Pyramid's threadlocal library to fetch it. This is one of the few places where it's recommended to use the threadlocal library.

15.12 Deployment

Deployment is the same for Pyramid as for Pylons. Use "paster serve" with mod_proxy, or mod_wsgi, or whatever else you prefer.

CHAPTER

SIXTEEN

APPENDIX: RANT ABOUT SCAFFOLDS AND PASTESCRIPT

The main reason the ‘akhet’ scaffold is gone is that maintaining it turned out to be a significant burden. Testing a scaffold requires several manual steps – change a line of code, generate an app, install it, test a URL, test some other URLs, change the application, backport the change to the scaffold, generate another app, install and test it, -OR- make changes directly to the scaffold and generate an app to see whether it works. If it requires custom application code to trigger the bug, you have to re-apply the code every time you crete the app. Beyond that, Pyramid evolves over time, so the scaffolds have to be updated even if they were working OK. And the scaffold API is primitive and limited; e.g., you can’t inherit from a scaffold and specify just the changes between yours and the parent.

The final barrier was Python 3. Other packages descended from Paste have been ported to 3 (PasteDeploy, WebOb), but Paste and PasteScript haven’t been. There doesn’t seem to be much point because the scaffold API needs to be overhauled anyway, many of paster’s subcommands are obsolete, and some people question the whole concept of plugin subcommands: what exactly is its benefit over bin scripts?

Pyramid 1.3 drops the Paste and PasteScript dependencies, and adds bin scripts for the essential utilities Pyramid needs: ‘pcreate’, ‘pserve’, ‘pshell’, ‘proutes’, ‘ptweens’, and ‘pviews’. These were derived from the Paste code, and the scaffold API is unchanged.

Two other factors led to the demise of the scaffold. One, users wanted to mix and match Akhet features and non-Akhet features, and add databases to the scaffold (e.g., MongoDB). That would lead to more questions in the scaffold, or more scaffolds, and more testing burden (especially since I didn’t use those databases).

The other factor is, I began to doubt whether certain Akhet features are necessarily better than their non-Akhet counterparts. For instance, Akhet 1 and Pyramid have different ways of handling static files. Each way has its pluses and minuses. Akhet’s role is to make the Pylons way available, not to recommend it beyond what it deserves.

So faced with the burden of maintaining the scaffold and keeping it updated, I was about to retire Akhet completely, until I realized it could have a new life without the scaffold. And as I work on my own applications and come up with new pieces of advice or new convenience classes, I need a place to put them, and Akhet 2 is an ideal place. So viva the new, scaffold-free, Akeht 2.

- *genindex*
- *modindex*
- *search*

The word “akhet” is the name of the hieroglyph that is Pylons’ icon: a sun shining over two pylons. It means “horizon” or “mountain of light”.

PYTHON MODULE INDEX